Measurement and Enhancement of Peer-to-Peer based File Synchronization with Cloud Assistance

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Ranga Reddy Pallelra

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Dr. Haiyang Wang

July 2015

Acknowledgements

I would like to take this opportunity to sincerely thank Dr. Haiyang Wang for his continuous support and guidance, without whom this thesis would not have been possible. I am greatly indebted to him for his invaluable suggestions throughout the work. Also, he has always taken care about the well-being of his students, I am immensely grateful to him for that.

I would like to thank Dr. Peter Peterson and Dr. Yang Li for serving on my thesis committee. A special thanks to Dr. Peter Willemsen who filled energy and enthusiasm in me, and for being my inspiration. Also, I thank Dr. Ted Pedersen for teaching me two wonderful courses Computer Architecture and Natural Language Processing. Further, I would thank Lori Lucia, Clare Ford, Jim Luttinen and International Student Services for their timely help.

I would like to thank all my friends in the class for all the fun, knowledge, encouragement, and for making my stay a memorable one. Thanks to Lavanya Singampalli for taking the time to proofread my thesis documentation and for her indispensable support. Lastly, I thank Jil Pavagadhi who stood by me and motivated me at all times.

Dedication

I would like to dedicate this thesis to my parents, Srinivasa Reddy Pallerla and Lakshmi Pallerla and my sister and best friend, Lavanya Pallerla for their everlasting love and endless support. In fact, I am very blessed to have such a family that has done everything for me. I also dedicate this thesis to every friend of mine who supported, motivated and inspired me in every walk of my life.

Abstract

Recent years have witnessed the rising popularity of file synchronization systems. Powered by rich datacenter resources, such commercial products as Dropbox and Google Drive not only provide conventional file hosting but also enable file synchronization with multi-party user collaborations. It is however known that their datacenter-based design will limit the system scalability. Peer-to-peer (P2P) based file synchronization, most notably BitTorrent Sync, is therefore widely suggested as a more scalable and efficient alternative. Unfortunately, the framework design and the protocol operation of P2P file synchronization remain vague to the general public. Identifying the exact performance bottlenecks or enabling theoretical and practical optimization is challenging and largely blinding to date.

In this thesis, we for the first time investigate the performance of P2P file synchronization in real-world measurement. We deploy BitTorrent Sync on highly-distributed PlanetLab testbed and closely investigate its framework design. Based on our measurements, we find that the P2P file synchronization system can provide very efficient file synchronization, especially for large contents. Different from traditional BitTorrent-like systems, our packet-level analysis also indicates that BitTorrent Sync does not have built-in tit-for-tat protocols. Peers' downloading speed is therefore, independent of their uploading. This modification naturally improves the synchronization efficiency yet also introduces certain fairness issues. For example, our measurement shows that 80% of our peers can obtain a 500MB file within 10 minutes. The remaining 20% users, on the other hand, will suffer from very long synchronization latency; the slowest peers will spend over 30 minutes to download the same file. To mitigate such a problem, we explore the potential to merge the existing P2P-based and cloud-based synchronization systems. The main idea of this hybrid framework is using cloud-based synchronization to accelerate the slower peers in P2P

synchronization. Using Dropbox and BitTorrent Sync as a case study, we implement this cloud-assisted P2P synchronization on PlanetLab testbed. The evaluation shows that the cloud-based enhancement can well-address the fairness issues while improving the overall synchronization efficiency by 38%.

Contents

List of Tables

List of Figures

1 Introduction

Internet-based file synchronization systems have achieved tremendous success in the contemporary IT industry. Such systems as Dropbox [11], Google Drive [15], and Microsoft OneDrive [24] not only provide online storage space but also enable efficient file synchronization for user collaboration. To understand the detailed framework of these systems, many studies have explored their design characteristics such as the cloud deployments [10] [20] as well the communication overheads [21]. Based on the existing measurements, it is known that such datacenter-based architectures will largely limit the system scalability. To mitigate this problem, the architecture of peer-to-peer(P2P) file synchronization has emerged as a successful alternative. However, its framework design as well as the protocol operation still remains largely unclear, that brings a significant challenge to pinpoint its potential performance bottlenecks.

In this thesis, we take a first step towards understanding the performance of P2P file synchronization system in real-world measurement. We deploy BitTorrent Sync on highly-distributed PlanetLab testbed over 50 nodes and closely investigate its protocol and performance. Our experiment shows that P2P file synchronization system is highly efficient especially for the delivery of large contents. This benefits from the removal of classic *tit-for-tat* peer selection mechanism in BitTorrent's framework. Unfortunately, our measurement further reveals that such a modification also introduces certain fairness issues. In a swarm with 50 peers, 80% of them can obtain a 500MegaByte file within 10 minutes. The synchronization latency of the remaining 20% users, however, can exceed 30 minutes without any bandwidth-related bottleneck. To obtain better fairness, we explore the potential

benefit of implementing a hybrid P2P file synchronization system with cloud assistance. In our case study, we use the popular cloud-based synchronization system, Dropbox to assist the peers in BitTorrent Sync. Our evaluation shows that the hybrid P2P-cloud design can largely enhance the synchronization of slow peers and improving the overall synchronization efficiency by 38%. For example, if we modify the previously discussed swarm with 50 peers, 80% of the peers can download the file within 12 minutes, and the remaining 20% will also be able to complete the downloading within 16 minutes.

The remainder of the thesis is structured as follows: In chapter- 2, we present the background and related works. Chapter- 3 discusses how BitTorrent Sync is configured so that measurements can be performed on PlanetLab to evaluate its performance. Chapter- 4 presents the measurement results and analysis of the same. In chapter- 5, the design of a new Hybrid-Cloud P2P file synchronization is demonstrated. Thereafter, its performance is evaluated by conducting experiments on PlanetLab. Some practical issues are also discussed. Chapter- 6 concludes the paper and provide details about future work.

2 Background

Large-scale data sharing and file synchronization techniques have gained tremendous popularity in the recent years. In this chapter, we will summarize the existing studies about content delivery and synchronization. These studies are from the conventional client and server systems to the emerging cloud and P2P systems such as Dropbox and BitTorrent Sync.

2.1 Client/Server File Delivery System

Client/Server (C/S) architecture is the earliest content delivery model on the Internet. As shown in Figure 2.1, the server in this framework is a centralized component. It provides such services as data storage and delivery to different clients. These clients, on the other hand, will need to initialize their requests for such services. The communication between clients and servers can build on different application layer protocols such as File Transfer Protocol (FTP), Hypertext Transfer Protocol (HTTP), Simple Mail Transfer Protocol (SMTP) etc.

File Transfer Protocol (FTP): It is a network protocol built on a client/server architecture used to exchange files over any TCP/IP based network. An FTP connection between a client and a server is made via port 21. A client that supports FTP is commonly used to download a file from the server or to upload a file to the server on the Internet.

Hypertext Transfer Protocol (HTTP): It is a request-response protocol designed to transfer multimedia files such as images, text, audio and video between client and server. Each transfer includes the client requesting a file from the server, and then the server replying with the requested file (or an error notification). In between the client and server there may be several intermediate nodes such as proxies, gateways, and tunnels. Although, HTTP communications usually take place over TCP/IP connections, it can be implemented on top of any other protocol on the Internet, or on other networks.

Typically, an HTTP client initiates a request via Transmission Control Protocol (TCP) connection to port 80 (HTTP port number is 80 by default) on a host. Upon receiving the client request, an HTTP server listening on that port sends back a status message such as "HTTP/1.1 200 OK", and the body of the requested file, or an error message if the file is not found. Resources to be accessed by HTTP are identified using Uniform Resource Identifiers through the http: or https (HTTP over Secure Socket Layer) URI schemes.

There have been numerous studies on the implementation, deployment analysis, and optimization on C/S-based CDN (Content delivery networks) (Pathan et al. [25]) and datacenter systems (Raiciu et al. [31]). In addition, the advancements in the field of mobile computing gave scope for Jing et al. [18] to perform comprehensive analysis of new paradigms such as extended client/server model, mobile client/server computing etc. Furthermore, Adler et al. [1] explored and implemented distributed client/server models that required tools such as remote procedure calls (RPCs) and message-passing systems for establishing and controlling communications between applications over the network.

Client/Server computing is a traditional paradigm which helps the clients to download the file content from the server. The advantages and disadvantages of client/server model

is presented in Figure 2.4. It is easy to see that the server capacity is a severe bottleneck in the C/S design. To address this problem, several solutions were proposed such as using large server farms(Gandhi et al. [13]), efficient load balancing infrastructures (Pinheiro et al. [26]) and content-caching proxies (Candan et al. [5]). However, the existing solutions are still not scalable in terms of the maintenance costs especially under flashcrowd user arrival (large number of users sends resource requests to the server creating high traffic on the network) (Van Eenennaam et al. [35]). A scalable content delivery model is therefore required to fulfill the demands of large-scale systems.

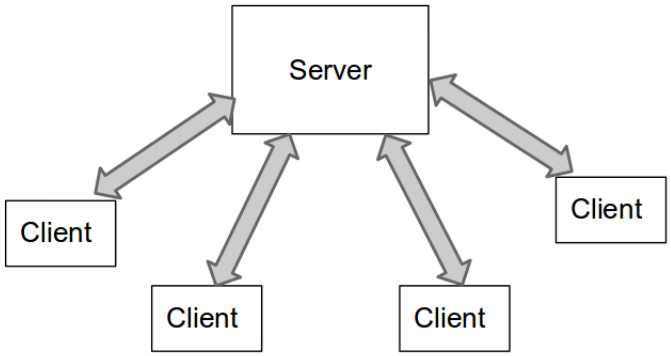

Figure 2.1: A server serving four clients.

2.2 Peer-to-Peer-based File Distribution

As a highly scalable framework, Peer-to-peer (P2P) content distribution have gained enormous popularity in recent years. Different from C/S-based file downloading, peers in P2P system work as servers as well as clients. A file downloaded by a peer is often made available for upload to other peers, forming a symmetric file distribution system. The primary advantage of a peer-to-peer file distribution network is that it is self-organizing and scales accordingly with the number of users initiating a download.

2.2.1 Overview of BitTorrent

BitTorrent is the most popular P2P file sharing system, introduced by Bram Cohen in 2001 [8]. It was mainly designed to distribute large files over the internet. It was immediately grasped by several content providers as a scalable method of distributing content, reducing the load on congested servers, diminishing the distribution costs and increasing the download rates for users. An interesting fact is that, large scale social networks such as Facebook and Twitter use the BitTorrent paradigm to push hundreds of megabytes of new updates to all their servers worldwide in an efficient manner. BitTorrent focuses on setting up an overlay network called a 'swarm' for all the peers who are interested in downloading the same shared content. In fact, as more users join a BitTorrent swarm, the more will be the downloading rate will proportionately increase for all the peers.

Any P2P design based file sharing application requires two functions to be supported. a) Search function: this allows peers to discover the content they are interested in, among other participating peers. b) Downloading function: allows peers to download the content upon locating it. Conventional P2P file distribution applications such as KaZaa and Gnutella [41] aims at quickly locating the peers that hold a given file. Upon locating the peers, the file is downloaded directly from them. On the contrary, BitTorrent works on the coordination of the peers for file distribution using the swarming technique [8]. BitTorrent varies from the traditional P2P file sharing applications in three viewpoints [29]. Firstly, it does not offer a search function. Rather, peers are relied upon the usage of central-directory based search techniques provided by BT websites. Secondly, it offers a file-level sharing mechanism instead of directory-level sharing mechanism. Lastly, it embraces a bartering mechanism such as 'Tit-for-Tat' among peers, which enforces a rule that, unless a peer adds to the continuous downloading of file chunks, it won't have the capacity to download the file.

BitTorrent Functional Elements

.torrent file: Prior to joining a BitTorrent network, a new peer needs to download the .torrent file from a BitTorrent portal. A .torrent file is a meta-data file that is associated to content shared through BitTorrent. The .torrent contains the following information: name of the shared content, content size, number and size of the chunks (content is divided into pieces named as chunks), torrent hash value(used to uniquely identify the BT swarm linked to the .torrent file) and IP addresses of the Trackers managing the peers in the swarm [19].

BitTorrent Portal: It is a server where content publishers upload .torrent files enabling the BitTorrent clients to download those .torrent files.

BitTorrent Swarm: When a set of peers is using the BitTorrent protocol to download the same content, they form a BT swarm.

BitTorrent Client: A peer who participates in a BT swarm by downloading and/or uploading chunks of the content is termed as BT client. The clients/peers are categorized into two types: (i) A client is termed as a seeder if it has a complete copy of the content, and acts as an uploader, uploading chunks of the content to the remaining peers in the swarm. (ii) A client is termed as a leecher if it does not have a complete copy of the content, and therefore both uploads and downloads chunks to and from other peers respectively in the swarm.

BitTorrent Tracker: It is a server that maintains the list of peers forming the BT swarm. Also, the tracker knows the download progress of each peer within the swarm.

Content Distribution in BitTorrent

In order to distribute the content through BitTorrent application, the content publisher creates a .torrent file associated with the content and then uploads it to a BitTorrent portal. (The BT content publishing methodology is analyzed in a detailed manner and the analysis

can be found at [9]). There are a few popular BT portals such as The Pirate Bay, Kickass Torrents, Torrentz etc. which host millions of torrents and thus receive millions of daily visits. These torrent websites or portals provide detailed information associated to each indexed torrent. The information offered by the portals, in general, include: category of the content, names of all associated files, size of the whole content in the torrent, uploading date, username of the torrent uploader, number of seeders and leechers contributing to the torrent swarm (this information is updated every few minutes) and others. Furthermore, it is worth knowing that a few of these popular portals offer a Rich Site Summary or Really Simple Syndication (RSS) feed to notify the newly published torrents.

Joining a BT swarm and Discovering peers

When a BitTorrent user wants to download a specific content A, the .torrent file specific to A is found in a BT portal which is downloaded by the user. The .torrent file can be accessed with any of the existing BT clients [40]. When the torrent file is opened with the BT client, it connects to one of the trackers included in the torrent file. Initially, a new peer sends the Tracker a announce started request, and the Tracker sends the number of seeders and leechers available in the swarm as a response. Along with that information, the Tracker also sends the IP addresses of more than 40 randomly selected peers. These peers act as the neighbors of the new node. At this stage, the new peer will be able to download the content from the swarm. If the number of neighbors of a peer falls below a threshold value of 20, it again contacts the Tracker with a new announce started request to obtain new neighbors. When a peer leaves the swarm, it sends an announce stopped request to the Tracker which removes this peer from the list of participants in the swarm.

BitTorrent Delivery Mechanism

In BitTorrent, the peer wire protocol facilitates the exchange of chunks of information between peers. Each communication between two peers starts with an initial handshake. After the handshake sequence, the peers exchange bit fields using a BITFIELD message. The bit field designates which pieces of the content a peer has already downloaded. Additionally, every time a peer gets a new piece, it notifies to its neighbors with the help of a HAVE message. Therefore, at any point of time each peer knows about the pieces that every other neighbor has in the swarm.

BitTorrent Algorithms

Choking/Unchoking algorithm: When the neighbors of a peer p have notified their interest in its contents, this algorithm determines the node among the neighbors to which peer p has to transfer the contents to. By default, all the connections are choked. If a peer wants to transfer the contents to another peer, it un-chokes the connection with that peer. Although a peer can upload content to multiple peers at a time, the number of current uploads are constrained to 5 by default [3]. Therefore, a peer in the swarm has 5 unchoked connections at any point of time. Out of these, four connections are selected based on "tit-for-tat" criteria.

A peer keeps track of the download rate from all its neighbors, transferring content to it. Among these neighbors, four with the highest download rates are unchoked. That means, a leecher rewards those other peers or leechers from whom it downloads more chunks within the last 20 seconds. The rest of the neighbors will be choked(blocked). However, in the case of a seeder, which has nobody to download from, the decision is made based on the upload rate to the neighbors. Hence, the seeder un-chokes four leechers to whom more chunks are uploaded in the last 20 seconds. So the leechers which download very quickly

from the seed get a higher preference. This decision to choke/unchoke a leecher is made periodically (for every 20 seconds) and depends on the download rate.

BitTorrent uses Tit-for-Tat method for the delivery mechanism, and this method biases traffic between peers toward the higher bandwidth routes. Basically, by this tit-for-tat method each leecher uploads chunks to those leechers from whom it downloads more chunks.

Apart from the regular unchoke operation used by BitTorrent, it also implements the optimistic unchoke operation in order to choose the last of 5 connections. With the help of optimistic unchoke operation, a leecher selects a neighbor randomly to unchoke, without considering the download rate from that peer. This unchoked node is chosen periodically once every 30 seconds.

Piece Selection When a leecher is unchoked by its neighbor, it can request a data block from that neighbor. The requested block is determined using the "Rarest First Policy". That is, since leechers have complete knowledge about the availability of all blocks in the neighborhood, they always request the rarest block or piece. Studies have shown that the optimal performance of BitTorrent is not dependent on this rarest first mechanism [3].

From the initial stages of BitTorrent development, several important limitations in the BitTorrent design have been addressed over time, and the changes are summarized in Table 2.1.

Firstly, only a single tracker was used while creating a torrent file associated to content that needs to be shared which raised an availability problem. To resolve this, the Multi-Tracker Metadata extension was introduced to support multiple trackers. Second, discovering of peers happened only through the trackers service, which raised anonymity concerns. The Distributed Hash Table (DHT) protocol was introduced to support the discovering and exchanging information between peers directly, creating a coarse global BT network. Third, even though peers connected through DHT protocol, they had no way of exchanging infor-

mation about the presence of other peers in the swarm. The Peer EXchange (PEX) protocol that supports this feature, enabling swarms to perform operations without the trackers was introduced. Lastly, to exchange content through BitTorrent protocol peers required the metadata information associated with the content, which was retrievable only from BitTorrent websites. With the extension for P2P Metadata, peers can exchange metadata among themselves, effectively creating a webless BT network.

Change(Year)	Effect
Multitracker Metadata (2003)	higher tracker availability
DHT Protocol(2005)	global BitTorrent network
Peer Exchange(2007)	trackerless BitTorrent
P2P Metadata(2008)	webless BitTorrent search

Table 2.1: Major changes in BitTorrent

2.2.2 Related Studies of P2P Content Delivery

There have been a lot of studies on the analysis, implementation, and enhancement on the peer-to-peer based content delivery systems such as BitTorrent. Some previous research efforts are put forth to understand the peer distribution of BitTorrent over the global Internet. Haiyang et al. [37] designed and demonstrated a novel hybrid PlanetLab experiment to interact with real-world BitTorrent trackers and peers. They found that the BitTorrent peers exhibit strong geographical locality.

Kryczka et al. [19] performed measurements to understand the BitTorrent ecosystem that analyzed the performance aspects such as the ratio of seeders/leechers, the session time of the BT users, the arrival rate of peers, estimating peers upload rates etc. The challenges faced while performing measurements and the possible solutions [19] to address these challenges were demonstrated. Androutsellis-Theotokis et al. [2] proposed a framework to analyze characteristics such as security, fairness, performance and scalability of peer-to-peer

content distribution technologies. Further, Magharei et al. [22] followed a performance-driven approach to design a system called PRIME for live streaming using mesh-based P2P technique. The system aims to mitigate the performance issues such as bandwidth bottleneck and content bottleneck. Carbunaru et al. [6] performed a PlanetLab-based measurement to identify different phases in peer bandwidth utilization while downloading a file in a P2P network. They focused particularly on analyzing the performance scalability of P2P file distribution and server provisioning during flash crowds.

The current implementations of Peer-to-peer applications such as BitTorrent do not take into consideration the underlying Internet topology or the traffic costs at ISPs and generate a large amount of cross-ISP traffic. The root cause of high cross-ISP traffic is that BitTorrent enables data transfers among randomly selected peers distributed around the Internet. Bindal et al. [3] implemented biased neighbor selection which is a new approach designed to improve traffic locality and thus reduces the cross-ISP traffic.

Figure 2.4 illustrates the strengths and weaknesses of peer-to-peer file distribution system briefly. The challenges encountered by P2P content delivery are explained below:

- The P2P system is not completely reliable because of high node churn rate i.e nodes arrive and depart anytime they want.
- In order to distribute a file on a P2P network, a user has to generate a torrent file, deploy own tracker server or use tracker websites and upload the generated torrent file to the online portals for publishing the content. This entire process is quite complicated and also not convenient for user collaboration/synchronization.
- In a dynamic and unreliable environment where peer churn rate is high, the maintenance overhead for the overlay P2P network is one of the major design concerns.
- There is a problem of malicious peers who attempt to corrupt, delete, deny access to

the file replicas which are distributed between nodes.

2.3 Cloud Based File Synchronization System

Cloud computing has become increasingly prevalent in recent years. Cloud users outsource their computation and storage to cloud service providers by utilizing their pay-per-use model. Contrasted with the traditional client/server computing model that uses dedicated and in-house infrastructure, cloud computing model offers phenomenal preferences in terms of cost and reliability. Cloud storage and file synchronization services, (for example, Dropbox, Google Drive, OneDrive etc.,) facilitates real-time storing and sharing of data making it more reliable and convenient from anywhere, on any device, and at any time. The user's data (e.g., documents, photos and music files) stored in cloud storage are automatically synchronized over all assigned devices such as PCs, smartphones and tablets) associated with cloud in a well-timed manner.

Large content providers such as YouTube and Netflix normally utilize a cloud-based model that relies on geographically dispersed content delivery networks (CDNs) to meet computation and storage demands of users. It is known that this new generation of file synchronization service is greatly benefited from cloud virtualization resources. This approach employs a huge and costly computing, storage and delivery infrastructure. For example, YouTube (subsidiary of Google), utilizes Google's own content delivery infrastructure, Netflix utilizes Amazon's cloud services and other third-party CDNs such as Akamai and Limelight, whereas Dropbox uses Amazon's S3 service for file storage.

In a few short years, cloud storage service providers such as Google and Microsoft have come to phenomenal levels of accomplishment, with their user base growing tremendously [20], thanks to their customers using Office and Gmail and Docs, respectively. Microsoft OneDrive claims that their user base has increased to 250 million while Google Drive has

obtained 240 million users as of September 2014 [39]. Despite its late entry into the market (April 2012), Google Drive had obtained more than 10 million users just in its first two months.

The primary function of cloud storage services is file synchronization which maps addition and modification of the user's data in the local file systems into the cloud through a series of network communications.

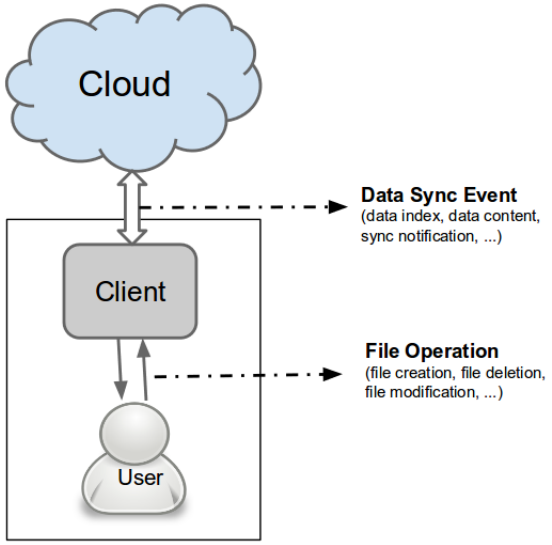

Figure 2.2: Data Synchronization Principle.

Figure 2.2 exhibits the general file synchronization principle. In a cloud storage service, the user usually has to create a local folder (called a "sync folder") in which all file operations (such as file creation, file deletion, file updation etc.,) are observed and synchronized to the cloud by client software developed by the cloud service provider. File synchronization process includes a sequence of data sync events such as data index, data content, sync notification, sync status, sync statistics and sync acknowledgment [20]. All the above specified data sync events generate network traffic which is termed as "data sync traffic".

2.3.1 Overview of Dropbox

Dropbox is one of the most popular personal cloud storage service providers. It has acquired more than 300 million users as of May 2014, who store or update 1 billion files every day. In addition to the file hosting, Dropbox offers to effectively share, edit, and synchronize online files. The key operations of Dropbox as a file synchronization service are detection and transmission.

Detection: The client must be able to automatically detect and update changes made, remotely on the cloud and on the local file system. Dropbox relies on push-based notifications to detect the remote changes, whereas to detect the changes on the local file system it uses the Linux's inotify service. To identify changes made while Dropbox is offline, the metadata information for each file is stored, which includes last modification and attribute change times, in a local database. While booting a device, Dropbox examines its monitored files, and uploads the metadata of any file if it has changed from its previous value.

Transmission: Dropbox splits each file into chunks of up to 4MBytes in size. When a user adds a file to his/her local Dropbox folder, the local Dropbox client will calculate the hash values of all the chunks of the file using the SHA-256 algorithm to avoid redundant file uploading. It then relies on Amazon S3 cloud storage service for storage purposes. While transmitting the changed files to and from the server, Dropbox utilizes a deduplication technique to reduce network traffic by calculating a hash value for each 4MB chunk in the file [42]. It sends the calculated hash for each chunk to the destination first and if the chunk is already present, it will not be sent. Furthermore, Dropbox relies on rsync utility to transmit only the modified portions of the chunk if it is partially changed. Finally, to accomplish the atomicity property at the client, Dropbox downloads chunks of files to a staging area, gathers them, and then moves them to their destination location.

Dropbox architecture comprises of two types of servers: control and data storage servers.

The control servers are under the direct control of Dropbox Inc., whereas Amazon Elastic Compute Cloud (EC2) and Simple Storage Service (S3) are used as storage servers. The Dropbox service framework consists of three major components on the server side (the gray boxes in the Fig. 2.3). 1) Load-balancers deployed by Dropbox. 2) Dropbox Delivery Servers that deploy tens of thousands of EC2 instances which provide data uploading, downloading and file processing functionalities. All user files are uploaded to these delivery servers during file synchronization. 3) Dropbox Storage Server like Amazon S3 server cluster is utilized to store the uploaded files. Hence, EC2 instances will serve as a bridge between client applications and S3 storage servers.

The steps followed during the data flow from a source user who wants to upload a file to his/her Dropbox folder are as follows:

1. A data source (user who has files to upload) sends a DNS request to query the IP addresses of the load-balancers in Dropbox. The DNS server responds with the list of load-balancers available to the data source. Then, the data source randomly chooses one of the load-balancers from that list and it transmits to the load-balancer the related metadata information of the file which includes the file size and type etc., The selected load-balancer is now assigned one EC2 server to the data source (Step A in the Fig. 2.3).

2. The data source uploads the file to the assigned EC2 server (Step B).

3. Upon successful uploading of the file, the EC2 server forwards this file to the S3 folders. (Step C).

4. Meanwhile, another EC2 server is used to deliver the file to all the destinations that need to be synchronized (Step D).

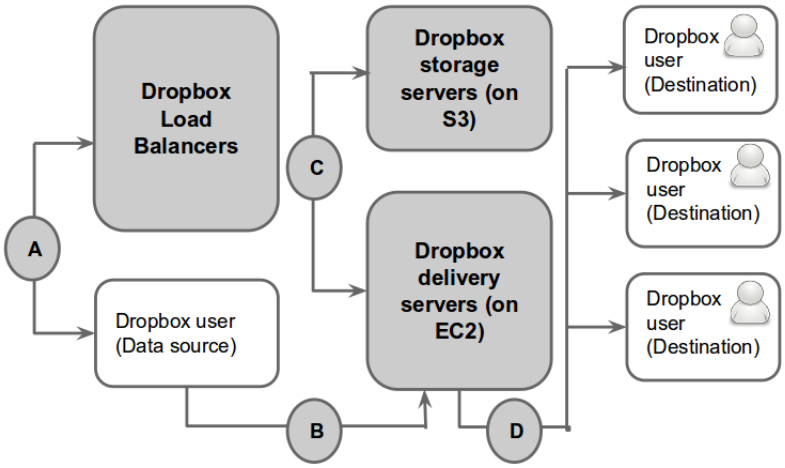

Figure 2.3: Dropbox Framework.

2.3.2 Related Studies of Cloud-based File Synchronization

A lot of effort has been put into understanding the behavior of cloud-based file synchronization services such as Dropbox, Google Drive etc. Gil et al. [14] presented a model to collect the data about the Dropbox usage through measurements and to understand its client behavior. Gracia-Tinedo et al. [16] demonstrated an active measurement study by comparing three cloud file synchronization systems such as Box, Dropbox, and SugarSync, to analyze the performance aspects like transfer speed and failure rate. Haiyang et al. [38] present a measurement to understand the design and performance bottleneck of Dropbox. Their measurements found that Dropbox not only relies on Amazon's S3 for file storage, but also uses Amazon's EC2 instances for providing file synchronization services. Therefore, they have proposed practical solutions to balance the bandwidth-intensive and CPU-intensive tasks on the virtual machines. The extensive peculiarity of Dropbox is presented by Idilio et al. [10], explaining typical usage, traffic patterns, and possible performance bottlenecks.

Casas et al. [7] evaluates the performance of personal Cloud Storage and File Synchronization applications such as Dropbox, Google Drive with the help of a group of 52 users and thereby analyzed Quality of Experience of such systems. Hu et al. [17] presents a measurement study by comparing four popular personal cloud storage services such as Dropbox, Mozy, Carbonite, and CrashPlan. They analyzed the performance attributes such as the backup and restore times, transferred data volumes, privacy risks, and backup faults etc.

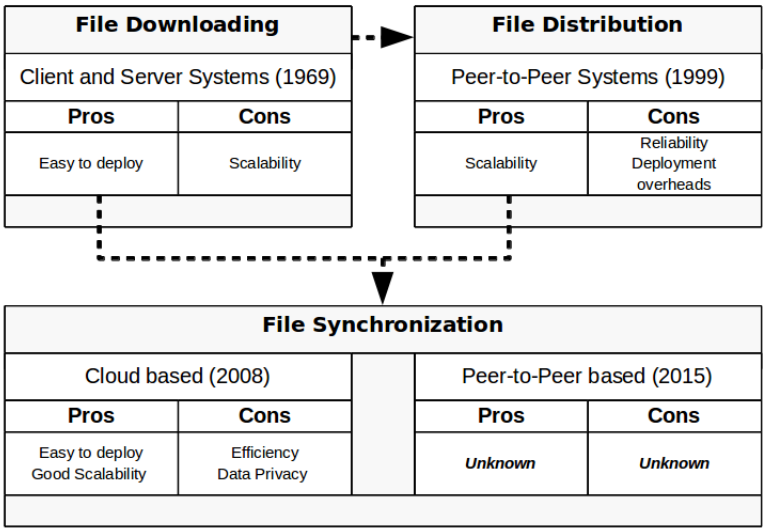

Figure 2.4: Transitions in File Delivery Systems.

Cloud-based file synchronization systems have advantages and disadvantages which are presented in the Fig. 2.4. These cloud-based systems are still facing challenges that need to be solved. They are:

- It is notified that files created on one device are synced to another device in a corrupt state [23].
- Even though cloud providers offer file storage and synchronization services at low

cost and with better scalability, the cloud-based deployment greatly increases the users' interaction latency.

- With the third-party storage services, there has always been data protection and privacy concerns for the users utilizing the cloud file synchronization services.

Figure 2.4 demonstrates the transitions evolved for distributing files from the file source to the destination users over time. Considering all the open problems discussed above, BitTorrent Sync, a P2P-based system is widely suggested as an alternative for file synchronization. There is no existing measurement study to understand the underlying functionalities of BitTorrent Sync application and hence the advantages and disadvantages of using P2P file synchronization are illustrated as unknown in the Fig. 2.4. Our work presents and explains the functioning, underlying features, protocols and typical usage of BitTorrent Sync, a P2P based file synchronization application. Scanlon et al. [33] presents the features of BitTorrent Sync application and proposes an investigation methodology which outlines the necessary steps involved in retrieving evidence from the network and analyzes the results. However, we performed measurements on this application to measure aspects such as synchronization latency, uploading/downloading speeds and CPU utilization.

3 Measurement Configuration of BitTorrent Sync

To mitigate the privacy and efficiency issues of cloud file, P2P-based file synchronization is widely suggested. Unfortunately, the framework design and the protocol operation of P2P file synchronization remain vague to the general public. Pinpointing the performance bottlenecks or enabling optimization is challenging and remaining largely unclear. In this section, we for the first time carry out a real-world measurement to understand the details of P2P-based file synchronization.

3.1 Framework of BitTorrent Sync

BitTorrent Sync [4] is the peer-to-peer file synchronization application developed by BitTorrent Inc. It is based on distributed technology, and there are no limitations on the data for syncing. The more devices you share data with, the faster the files get synced [4]. Since it uses peer-to-peer methodology, cloud usage is not required which means your files are not stored on third-party servers. The complete data set transferred using BTSync resides on at least one of the synchronized devices. This leads to much simpler data detection for digital forensics purposes as such it is not necessary to contact a cloud service provider to get the traffic to and from a container using authorized credentials.

BitTorrent Sync (also referred as BTSync) provides synchronization functionalities [12] similar to that of cloud file storage services such as:

1. **Synchronization Options:** Like Dropbox, BTSync also offers the service of syncing the user's content over LAN or over the internet.

2. **Availability:** BTSync is compatible with some of the prevalent desktop and mobile operating systems such as Linux, BSD, Mac OS, Windows, Android, iOS and Fire OS.

3. **Peer-to-Peer Technology:** The files are transmitted in a decentralized fashion among peers over the internet.

4. **Encrypted Data Transmission:** While sharing the files among peers, each file is broken down into chunks at the source. These chunks are encrypted using AES-128 encryption technique in the network transit. They are decrypted and reassembled when they arrive at the destination.

5. **No Limitations:** Unlike all the cloud file synchronization services, BTSync puts no limitations on the amount of data that needs to be synced. The reason is that the user's data is never uploaded to any third party server and hence storage volume is only limited to the user's own hard drive.

6. **Automated Syncing:** After the initial installation of the application and configuration, the data copied into selected folders gets automatically synchronized between machines.

As a consequence of the above alluring properties, BTSync has developed to turn into a well-known distinct option for cloud synchronization services. The innovation had developed to more than one million users by November 2013 and has multiplied to two million clients by December 2013. The services will without a doubt be of enthusiasm to both law requirement officers and digital forensics investigators in future examinations. While

BTSync is based on the same technology as BitTorrent for the file sharing, the aim of the application is truly distinctive. This outcomes in a change of users behavior and an essential change in the assumptions an investigator ought to make. BitTorrent is intended to be a one-to-many data distributing utility. The uploader ordinarily does not think about the downloader's identity and a single seeder can deliver data to an extensive number of one of a kind peers over the life of the torrent file. While the data is in transit, data integrity, and download speed are the top priorities over data privacy. Whereas BTSync on the other hand, is intended to be a safe and secure data replication protocol for making a dependable replica of a data set on a target machine. Data integrity is still highly considerable aspect, however, data privacy is currently the top need and speed-through-distribution is relinquished thus [33]. The files can only be read by users particularly who are offered the access to the repository. The notification of data availability is totally scalable by the owner with choices ranging from confining access to known IP addresses through registration with a centralized tracker.

Sync App Specifications

All the specifications [4] described below are provided by BitTorrent, Inc.

Shared Secret Keys: BTSync uses secret keys to uniquely identify between the shared folders. The secret keys are 20-byte strings that are human readable generated by using the random functions with /dev/random (on Mac and Linux) and Crypto API(on Windows). They are displayed using Base32 encoding technique. BTSync supports three categories of secret keys displayed in the Fig. 3.1 in order to share the data contained within specific folders.

1. **Master or read/write secret:** Any user who gets the access to this type of secret

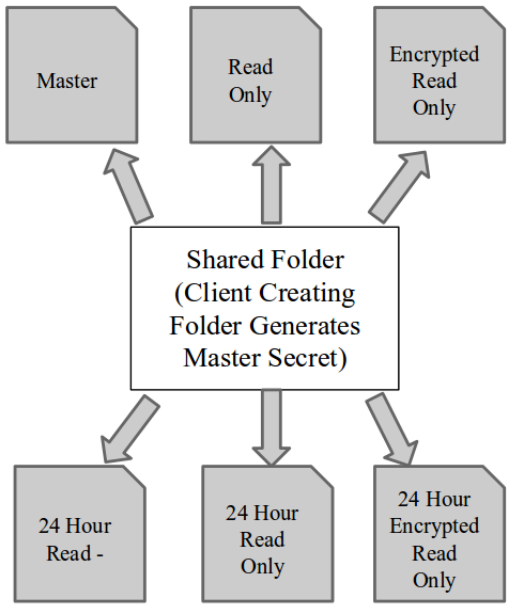

Figure 3.1: Categories of Secrets available for each Shared Folder.

is granted both the read and write access to the shared folder. For instance, every user utilizing this secret has the capacity to add files to be replicated over all other machines. Upon receiving the full access with this secret for any client, the access cannot be withdrawn without the creation of a completely new shared folder (thus starts replicating a fresh copy among the shared nodes). However, if a machine has downloaded the shared content, it's not possible to remotely delete that content. By default, if a content is remotely deleted, this deletion is reflected on each machine where the content is shared but the deleted content is copied into the .SyncArchive folder and is stored for 30 days before permanent deletion. Upon the initial creation of shared folder based on the version of BTSync, the 20-byte generated secret is prepended with the character 'A' or 'D' to form a 42-byte secret for distribution.

2. **read-only secret:** This secret permits hosts to share the folder in a one-way synchronization which is just the read access to the folder. Based on the version of BTSync,

the 20-byte generated secret is prepended with the character ‘B’ or ‘E’.

3. **encrypted read-only secret:** This type of secret allows remote machines to synchronize an encrypted replica of the shared content. The remote machine will have only read access to the content. Only the recent version of BTSync has the ability to generate this secret which begins with the character ‘F’.

For all the mentioned above three categories, the generated secret is always linked to a specific shared folder with the predetermined level of access. Nevertheless, the user can likewise create a time sensitive variant for each of those secrets as indicated towards the base of Figure 3.1. This variant specifies that joining the shared folder is admissible only for 24-hours. Upon the completion of 24-hour time frame, the secret will get to be invalid. However, the users who already have added this secret to their BTSync installation will keep on having admittance to upgrades from shared folder. Only the master secret (read-/write) or read-only secret will be recorded in the sync.dat file, but not the 24-hour secret.

Protocols: Sync uses TCP/IP and µTP [4] for transferring the data over the network. µTP is a P2P protocol specifically designed for sharing large files across multiple devices.

Connectivity: The file transfer is initiated after the connection between the devices is established. The connection is made using UDP, NAT traversal and UPnP port mapping. Relay and tracker servers are setup to make certain that the devices are connected. Relay server is used if the devices are not connected directly due to NAT issues. If the devices belong to the same local network, the connections can be made using LAN instead of using the internet for faster synchronization.

Peer Discovery: The Sync app does this in five ways. They are:

- **Local peer discovery:** To find out the peers located in the same local network, Sync sends the broadcast packets. Peers respond to those broadcast messages if they are available and get connected.

- **Known Hosts:** Specify the static ip:port in the Sync client configuration file to get connected to the peers using this information.

- **BitTorrent Tracker:** All the available peers are discovered using this tracker server. Also, the connection is facilitated by the tracker. These tracker servers maintain a list of seeders (peers that have complete file and upload the file to share with the other peers) and leechers (peers that do not have the file and are in the process of downloading the file from the swarm). These peers send status reports to the tracker server periodically and the server updates the list of active peers.

- **Peer Exchange(PEX):** Initially, peers are only allowed to transfer data content between each other and in order to locate other peers they had to depend on the tracker servers. To reduce this dependency on the servers, with the help of PEX protocol the connected peers exchange the information of other peers that are currently available in the swarm.

- **Distributed Hash Table(DHT):** Sync uses DHT to send its information and also to acquire other peers information.

Synchronization process: When a new file is added into the shared folder or when an existing file in the shared folder is modified, it triggers the file synchronization process. If the file is having the extension .Sync at the destination, then it's currently being downloaded or updated. File updates come by patches with the .SyncPart extension. If at all a file is

deleted on the source device, it will be stored in .SyncArchive folder which is a hidden directory in the sync folder of the syncing device.

BitTorrent gathers some information from the users which can help them to analyze the performance of the application. The information is about:

1. SHA2(secret) on the tracker, to connect the peers

2. The amount of data synced

The framework design and protocol operation of BitTorrent Sync is unknown to the general public. Also, since BTSync is a peer-to-peer based file synchronization tool, performing measurements on it is quite challenging and hence there is no much research to understand its performance. Therefore, we performed real-world measurements to understand the application and to identify its performance bottlenecks. The main contribution of this thesis work is to investigate the performance of BitTorrent Sync file synchronization protocol. We find the design and performance issues of BTSync by analyzing our real-world measurements.

3.2 PlanetLab-based Active Measurement

In this section, we present the design of our measurement approach. In particular, we illustrate the details of the tools used for performing measurements mainly focusing on PlanetLab and related scripting, and analysis of code. Python and Bash scripting languages were used for the measurements. Python scripts are run on a local system that triggers the Bash scripts on the selected remote PlanetLab nodes in order to measure all the performance metrics such as upload/download rates, CPU utilization and synchronization latency. Further, all the results from these remote nodes are retrieved and stored on the local system.

3.2.1 PlanetLab

PlanetLab is a global network testbed [28] which provides the users a platform to run network-based and distributed applications across plenty of nodes throughout the world. It is a large group of systems spread around all the continents purely for research purposes. The researchers conduct real-time experiments which look more realistic rather than a simulation.

Terms related to PlanetLab Site: It is a physical location where PlanetLab nodes are located. e.g: University of Minnesota. If you have an account then the details of your site can be displayed on the PlanetLab website when you click on the tab "My Site" under the section "Sites"

Node: Each site provides, either one or more dedicated servers termed as "nodes" to the PlanetLab network. In return, the university is allowed to use all the PlanetLab nodes for its research purposes. There are 1353 nodes across 699 sites worldwide [28], as illustrated by Figure 3.2. Any authorized user can access all the PlanetLab nodes through his slice to run the experiments.

Slice: It is a collection of resources which basically is an access to the nodes. A slice is a kind of virtual machine which is isolated from the other slices. Each slice is provided certain resources such as CPU consumption, RAM memory, disk space, inbound and outgoing bandwidth, a number of connections. A user is assigned a slice and then any number of nodes can be added to the slice by the user to run the experiments.

Sliver: It is a collection of allocated resources on a single node.

Virtualization: Instead of provisioning the physical machines to users, PlanetLab heavily uses virtualization technology and provide users with virtual machines. Slivers are currently implemented as LinuxContainers (LXC) [28], which is a container-based OS-level

virtualization mechanism. LXC implements both namespace and performance isolation among all the slices. Previously, Linux-Vserver was used as node provisioning mechanism for all the PlanetLab nodes. Now, PlanetLab has been migrated to LXC which is a fast and robust virtualization mechanism in the Linux kernel. Currently, half of the PlanetLab nodes run on Fedora 14 (Laughlin) operating system and the remaining half run on Fedora 8 (Werewolf). Virtualization allows the applications to share physical resources and helps the applications or services to run for a long period of time.

Figure 3.2: Map of PlanetLab node Locations.

Usually, a normal application doesn't exceed the PlanetLab's Acceptable Use Policy (AUP), but in case the research application needs extra resources, it is possible to increase the limits (for example, if an application utilizes all the available physical memory and when the swap space is almost exhausted, PlanetLab kills the slice on that particular node and suspends it until the owner provides an explanation). Also, the slice expires after a given time period, and to avoid the expiration, the slice must be refreshed, with a renewal interval of 1 month.

To log into the slice, the user has to generate an RSA key pair and upload the public key to his/her PlanetLab account in the "key management" section under "My Account"

section. Upon finishing the upload, it may take up to 48 hours for the user to log into the PlanetLab nodes using the SSH protocol. Once a user gets logged into a PlanetLab node, he/she will have root access to the resources on that node. The command used to connect to the planetlab1.dtc.umn.edu node on the umn_haiyang slice is shown below:

```
# ssh -l umn_haiyang -i ~/.ssh/id_rsa planetlab1.dtc.umn.edu
```

Python Code for Connecting and Authenticating a PlanetLab user PlanetLab provides a programmatic interface XML-RPC for users to easily access and manage their applications and computing resources. The Python module authenticate() creates an object api_server whose methods help in validating the user's PlanetLab account can be seen in listing 3.1.

Retrieving information from PlanetLab nodes

The GetNodes() method is used to retrieve information about PlanetLab nodes, which takes an authentication structure as its first and main parameter. The remaining two parameters are optional which help in obtaining the filtered information, and if these parameters are not provided, GetNodes() returns all the details. The obtained information is a list of all node structures, and each node structure contains several named fields such as node ids, host names, authority, node status, number of CPU cores and other status information. The below code segment returns a list of all node details.

```
1 all_nodes = api_server.GetNodes(auth)
2 print all_nodes
```

If the second parameter provided for GetNodes() method is a list of node IDs or hostname

Listing 3.1: Python module to validate a user's PlanetLab account

```
import xmlrpclib

def authenticate():
    api_server = xmlrpclib.ServerProxy('https://www.planet-lab.org
    /PLCAPI/', allow_none=True)

    '''The first parameter to each XML-RPC call is an authentication
       structure. The code below shows how to set up this structure
       for password-based authentication.
    '''
    #Create an empty dictionary (XML-RPC struct)
    auth = {}
    # Specify password authentication
    auth['AuthMethod']='password'

    # Username and password
    auth['Username'] = 'sample.email.address@host.com'
    auth['AuthString'] = '********'

    '''Now we can verify this structure with the PlanetLab Central
       (PLC) API method AuthCheck(), which returns 1 if the
       authentication structure is valid.
    '''

    authorized = api_server.AuthCheck(auth)

    if authorized:
        print 'You are authorized to use PlanetLab!'

    return (api_server, auth)
```

of particular nodes, it will return information only about these nodes.

```
# Get information about two nodes at University of Minnesota.
minnesota_nodes = api_server.GetNodes(auth,
      ['planetlab1.dtc.umn.edu', 'planetlab2.dtc.umn.edu'])
print minnesota_nodes
```

Instead, to obtain the node information with specific fields we can specify using the second parameter, as such GetNodes() will return only those nodes with matching fields. Furthermore, the third parameter specifies which named fields to return. The below code snippet illustrated, retrieves only node IDs and hostnames of nodes with boot state as “boot”.

```
# Get node IDs and hostnames of nodes whose boot state is
#"boot"
boot_state_filter = {'boot_state': 'boot'}
named_fields = ['node_id', 'hostname']
nodes_with_boot_status = api_server.GetNodes(auth,
  boot_state_filter, named_fields)
print nodes_with_boot_status
```

The above demonstrated python code snippets retrieve information about all the available PlanetLab nodes. Instead, to obtain the information only about the nodes associated to your slice,the GetSlices() method must be used. The first parameter of GetSlices() is same as that of GetNodes() and the second parameter takes the user’s slice name. Listing 3.2 prints all the nodes associated to the slice “sliceName” with boot status and prints only the hostnames of every node.

3.2.2 Parallel Node Control via Vxargs

It is a time consuming and a repetitive task to run the same commands on multiple PlanetLab nodes. Vxargs [36] is inspired by xargs (Unix command-line utility) and a parallel version of the openssh tools such as pssh. It is a python application which makes it easy for the users to perform parallel execution of commands on a certain number of nodes, provided as either IP addresses, or hostnames, or both. Vxargs helps in monitoring a large set of machines over a wide area network, and also provides a real-time visualization of

Listing 3.2: Python code that prints all the nodes with boot status

```
slice_name="sliceName"
#Get the node ids that are assigned to the slice
node_ids = api_server.GetSlices(auth, slice_name,
                    ['node_ids'])[0]['node_ids']

#Get the hostnames that are assigned to the slice
node_hostnames = [node['hostname'] for node in
               api_server.GetNodes(auth, node_ids, ['hostname'])]

#get the complete information of each node which is assigned to
#the slice
node_info = api_server.GetNodes(auth, node_hostnames)

#boot_nodelist has all the list of nodes with the boot status
for node in node_info:
    if node['boot_state'] == 'boot':
         print node['hostname']
```

the execution of commands and its complete report. The commands will be executed on specified remote PlanetLab nodes parallely, that would otherwise require a serial execution which consumes a lot of time.

Vxargs can run any shell command on the remote nodes; commands often executed in PlanetLab environment are ssh, scp, rsync [32]. Vxargs takes at least two arguments to execute: a text file containing the list of nodes and the command to run. We can introduce a certain number of threads using -P argument. Each thread runs the given command individually on each node. Vxargs is run from the prompt and a text file with the list of IP addresses or hostnames of the PlanetLab nodes is passed with the help of -a argument. Another feature of vxargs is redirection i.e the standard output/errors of each individual job are redirected to files respectively for further analysis using -o argument.

For example, to execute the whoami command on every planetlab node in “allNodes.txt” file which has IP addresses of all the nodes, use the following way:

```
# python vxargs -t 20 -a allNodes.txt -P 10 -o Results ssh
```

```
slice_name@{}    'whoami'
```

In the above command {} variable is substituted dynamically with the IP addresses of nodes present in the "allNodes.txt" file and executed with 10 threads and the standard output is stored in the output folder named as 'Results'. If there are more number of nodes in the input file then each thread after completing its task takes another node from the input file and runs the job. The standard output/error is generated for each node in a separate file named after the planetlab node and saved in the 'Results' folder. The output files are really helpful since it is able to determine which nodes failed and also the reasons for their failures. When some nodes do not work or when they are slow, timeout option(-t) can be used in freeing the thread and assigning another node to it.

Vxargs Standard Ouput: The standard output folder contains three files after execution of a command via vxargs: host.out, host.err and host.status. host is the hostname or ip address of the node which is exactly as it appeared in the input file provided to vxargs. For example, when a command is executed using vxargs on node planetlab1.dtc.umn.edu, the output folder contains the files planetlab1.dtc.umn.edu.out, planetlab1.dtc.umn.edu.err, and planetlab1.dtc.umn.edu.status. The output from the remote node is stored in the .out file, and if the command has failed executing because of some error, the error message is stored in .err file. Further, the third file .status contains the exit status value of the command executed by vxargs (a positive integer). There is also a file "abnormal_list" created by vxargs, contains the list of nodes that failed to execute the command.

3.2.3 PlanetLab-based Deployment of BTSync

Upon installing BTSync on PlanetLab, it has to be configured accordingly to perform the file synchronization process. Executing the below command runs the BTSync as a background process with the default configuration.

```
# ./btsync
```

Instead, we can run the btsync application with our own configurations. Initially we would require a sample configuration file of the application. BTSync comes with an option to generate a sample configuration file by using the following command.

```
# ./btsync --dump-sample-config > sync.conf
```

The above command saves the sample configuration which is in JSON format into "sync.conf" file and it can be seen in the listing 3.3. Now we can make our own configuration file by editing the above generated sample configuration. The first field "device_name" has to modified to "slice_namenode_ipaddress" where slice_name is the user's slice name and node_ipaddress is the IP address of PlanetLab node on which BTSync is executed. If "listening_port" field value is 0, the port number is randomly chosen by the application at runtime. Instead, we can provide a unique and unused port number for the service to listen. The next step is to uncomment the "shared_folders" field completely that enables the application to share the folder. A folder has to be associated with a secret key that needs to be synchronized with the destination node. BTSync provides two options for creating a secret key. 1) The default secret for a folder (that needs to be synced) gives full permissions to the linked destination device on that folder. Upon the synchronization, any changes on a linked folder will be replicated in the source folder and across all other linked folders. Following is the command to generate a unique 32 character secret:

```
# ./btsync --generate-secret
```

Listing 3.3: Sample BitTorrent Sync Configuration file

```
{
  "device_name": "My Sync Device",
  "listening_port" : 0, // 0 - randomize port
/* storage_path dir contains auxilliary app files if no storage_path
   field: .sync dir created in the directory where binary is located.
   otherwise user-defined directory will be used */
  "storage_path" : "/home/user/.sync",
// "pid_file" : "/var/run/btsync/btsync.pid",
  "check_for_updates" : true,
  "use_upnp" : true,   // use UPnP for port mapping mapping
/* limits in kB/s: 0 - no limit */
  "download_limit" : 0,
  "upload_limit" : 0,
/* remove "listen" field to disable WebUI
remove "login" and "password" fields to disable credentials check */
  "webui" :
  {
    "listen" : "0.0.0.0:8888",
    "login" : "admin",
    "password" : "password"
  }
/* !!! if you set shared folders in config file WebUI will be
   DISABLED !!! shared directories specified in config file
   override the folders previously added from WebUI.*/ ,
  "shared_folders" :
  [
    {
//  use --generate-secret in command line to create new secret
      "secret" : "MY_SECRET_1", // * required field
      "dir" : "/home/user/bittorrent/sync_test", // * required field
//  use relay server when direct connection fails
      "use_relay_server" : true,
      "use_tracker" : true,
      "use_dht" : false,
      "search_lan" : true,
//  enable SyncArchive to store files deleted on remote devices
      "use_sync_trash" : true,
//  specify hosts to attempt connection without additional search
      "known_hosts" :
      [ "ipaddress:44444" ]
    }
  ]
}
```

The generated secret (For example: ABCDE0FGHIJK1LM2NOPQ7RSTUVW8XYZ) contains alphanumeric string with uppercase alpha characters. 2) Alternatively, only read permissions can be provided using a read-only secret, which means that any changes on a linked remote folder will not affect the source folder or any other linked folders. To generate a read-only secret, we need to pass the secret we generated using the default way to the following command:

```
# ./btsync --get-ro-secret ABCDE0FGHIJK1LM2NOPQ7RSTUVW8XYZ
```

The above commands generates another alphanumeric secret which is slightly longer and that permits read-only access to the folder. Upon generating the secret, paste it in the "secret" field in the config file. Finally, under the "known_hosts" field insert the PlanetLab node's IP address and the port number in the format of "IP:port_number". The remaining fields can be left same as in the sample configuration or can be modified accordingly. With these above modifications, we have configured the BTSync application which helps in syncing the folders with other remote PlanetLab nodes. Now, after modifying the configuration file sync.conf, using the following command we can run BTSync:

```
# ./btsync --config sync.conf
```

Upon configuring the BTSync application, we deploy it on PlanetLab (PL) and perform the required measurements. The steps followed in order to perform the measurements on two PlanetLab nodes (source and destination) are demonstrated below and Fig 3.3 presents the flowchart for the same.

1. Initially, the PlanetLab account has to be authenticated using PlanetLab Central(PLC) API. That means, username and password of the account are to be validated to use its

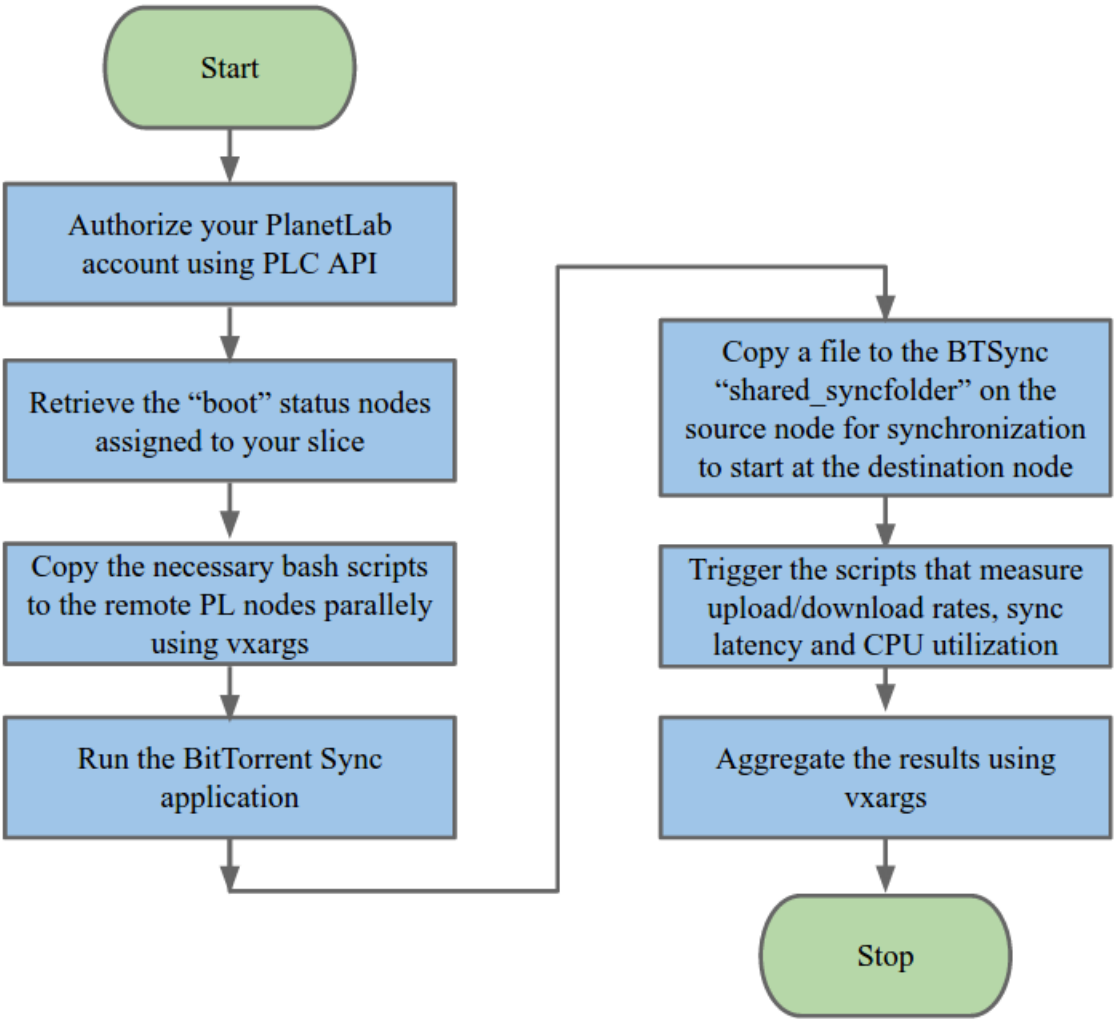

Figure 3.3: Flow chart for the framework to perform measurements on PlanetLab.

resources.

2. All the working PL nodes assigned to the slice with the status as "boot" can be chosen for running the measurements.

3. The BTSync application files and other necessary bash scripts are copied to selected remote PL nodes simultaneously using vxargs. The following command is used to securely copy the files on to remote nodes:

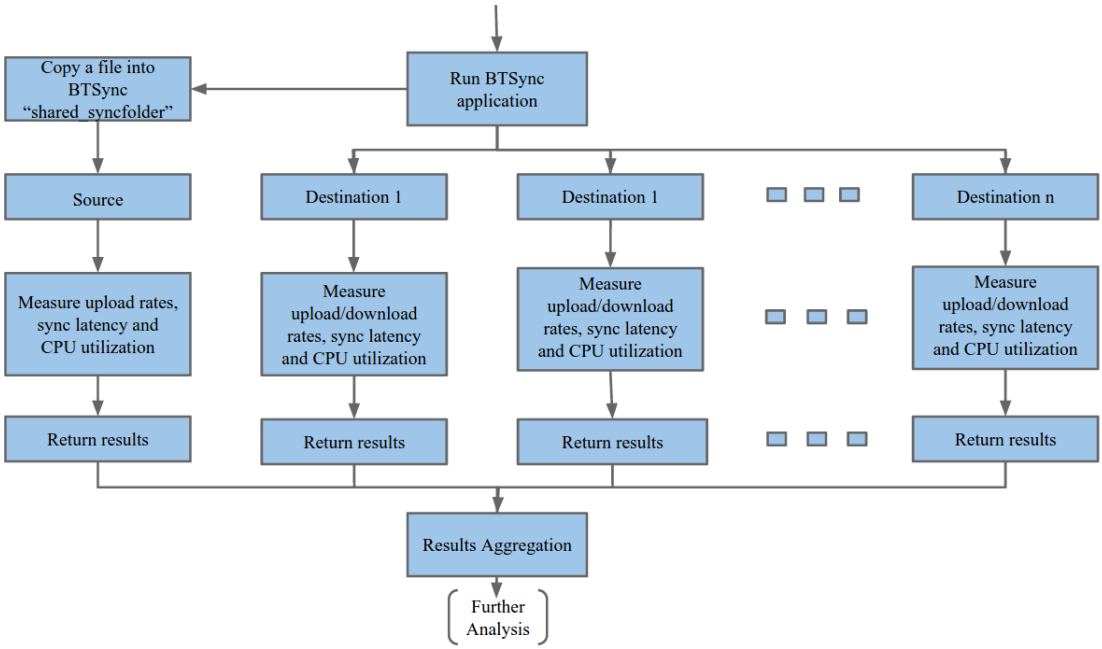

Figure 3.4: Overview of framework to perform measurements on multiple nodes.

```
# python vxargs -t 20 -P 10 -a nodes.txt scp -o
stricthostkeychecking=no btsync_scripts/* slice_name@{}:~
```

All the files from "btsync_scripts" directory are copied using 'scp' (SSH Secure Copy) command on to remote nodes specified in the input file "nodes.txt"

4. Upon copying the necessary files to the remote nodes, run the BitTorrent Sync application on the nodes.

5. Now generate a file of specific size using "dd" unix command-line utility, and move it to the BTSync "shared_syncfolder" on the source node such that the file synchronization begins via BTSync application at the destination node.

6. When the generated file is moved to the "shared_syncfolder", execute necessary bash scripts in order to capture the performance metrics such as upload/download rates,

synchronization latency and CPU utilization. For example, the following command is used to run a script to capture the upload speeds on the remote PlanetLab nodes:

```
# python vxargs -P 10 -t 50 -o Results -a nodes.txt ssh
-o stricthostkeychecking=no -l slice_name -i ~/.ssh/id_
rsa {} 'bash upspeedCapture.sh'
```

7. The results produced on each node is aggregated on the local system using vxargs.

Measurements are also conducted on multiple PlanetLab nodes to precisely understand the performance bottlenecks of BitTorrent Sync file synchronization application. The system implemented for this purpose is presented in the Fig. 3.4. The initial four steps are same as in the flowchart illustrated in the Fig. 3.3. To perform measurements, the system is implemented in such a way that the source and multiple destinations nodes are randomly selected using a Python random function. On all the destination nodes the bash scripts are executed to find upload/download rates, synchronization latency and CPU utilization whereas the source node does not require to download the file since it is the owner of the file that is being distributed among the peers. Therefore the download rates are not captured on the source node. Upon executing the scripts on all the nodes, the results of each node are accumulated on the local system and further analyzed to understand the design and performance issues of BTSync application.

3.3 Practical Issues

The problems encountered during running the measurement on PlanetLab were mainly: 1) PlanetLab runs on Fedora 8 and Fedora 14 which are pretty older versions (latest version is Fedora 22), and hence we could not be able to get the updated packages. Our task would have been much easier if one of recent Fedora versions was installed on PlanetLab. 2) Due

to limitations on Bandwidth [27] and Physical Memory, at times our processes were killed by the PlanetLab node administrators when the limitations were exceeded. 3) At some point of time, few PlanetLab nodes were down due to maintenance purposes and for some other reasons. 4) Python multithreading does not allow multiple threads to execute at the same time because of Global Interpreter Lock (GIL) issues [30] and hence we have migrated to Python Multiprocessing technique.

4 Measurement Results and Analysis

This chapter describes details of experiments performed to analyze the performance of Peer-to-peer file synchronization application BitTorrent Sync. The experiments were conducted on the PlanetLab, a computer networking research testbed for deploying and running distributed applications. The hardware configuration every PlanetLab node is 2.67GHz CPU with 4 cores, 4 GB memory and a bandwidth of 10 Mbps. This thesis focuses on understanding the attributes of BitTorrent Sync application such as downloading/uploading rates, synchronization latency, and CPU utilization. The results obtained from the experiments are presented in the form of graphs and analyzed.

4.1 Understand BitTorrent Sync Protocol

In order to understand the working of BitTorrent Sync, we run a simple experiment on two PlanetLab nodes. For this purpose, BitTorrent Sync application is deployed on these nodes. Upon deploying the application, we begin the experiment by syncing the content between these two BitTorrent Sync users (PlanetLab nodes). One user is considered as source who shares the file with the destination user who can be able to synchronize the shared file. The data source node is located at the University of Wisconsin - Madison, USA and the destination node is located at Max Planck Institute for Software Systems, Germany (nodes are randomly chosen).

To synchronize a file from the source node to destination node, the file content should be placed in the shared folder of BitTorrent Sync application. The file automatically starts

syncing between the two nodes only after the secret key of the folder at the source is used at the destination. Also, if the file is modified in the shared folder, the syncing starts directly after the file is saved with all the modifications. Once the synchronization process gets started at the destination, a temporary ".!Sync" file is created by the application. When the file is completely downloaded, the ".!Sync" file is now renamed with the original file name by removing the .!Sync extension at the destination. For example, a file named "testfile.txt" is uploaded at the source side and at the receiver side, the file begins downloading with the filename as "testfile.txt.!Sync". When the file is fully downloaded, it is then renamed back to "testfile.txt".

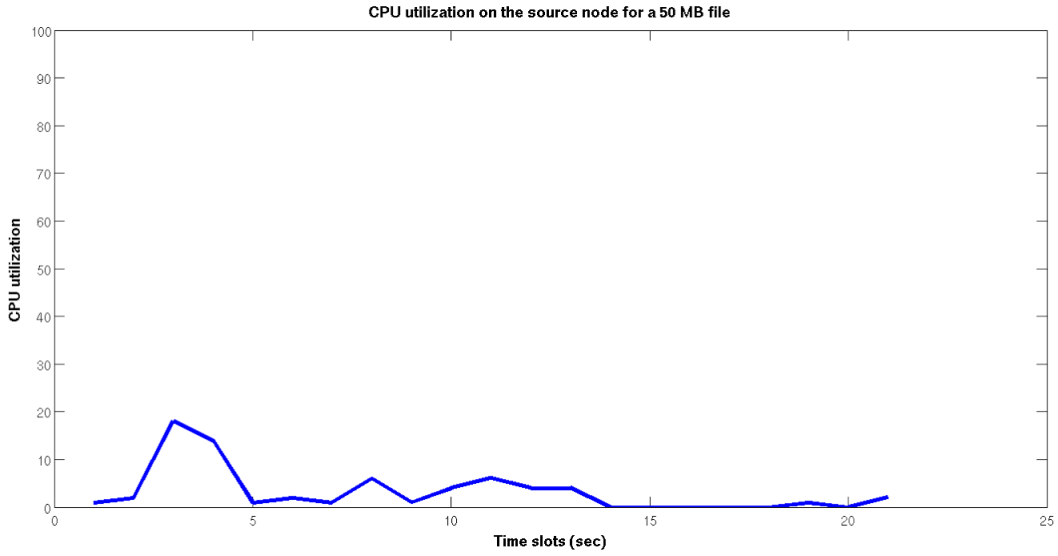

Figure 4.1: CPU utilization on the source node while synchronizing a 50MB file.

In the initial 10secs when a file was placed in the shared folder at the data source node, the CPU utilization elevates. This is a pre-processing step that occurs at the source node which is the cause for the increase in CPU usage. In this step, the file is split into several chunks and their hash values are computed. The above discussion is about the scenario where the file synchronization occurs between a source and a destination. As we can see in the Figure 4.1 the CPU utilization at the source node increases initially and then gradually

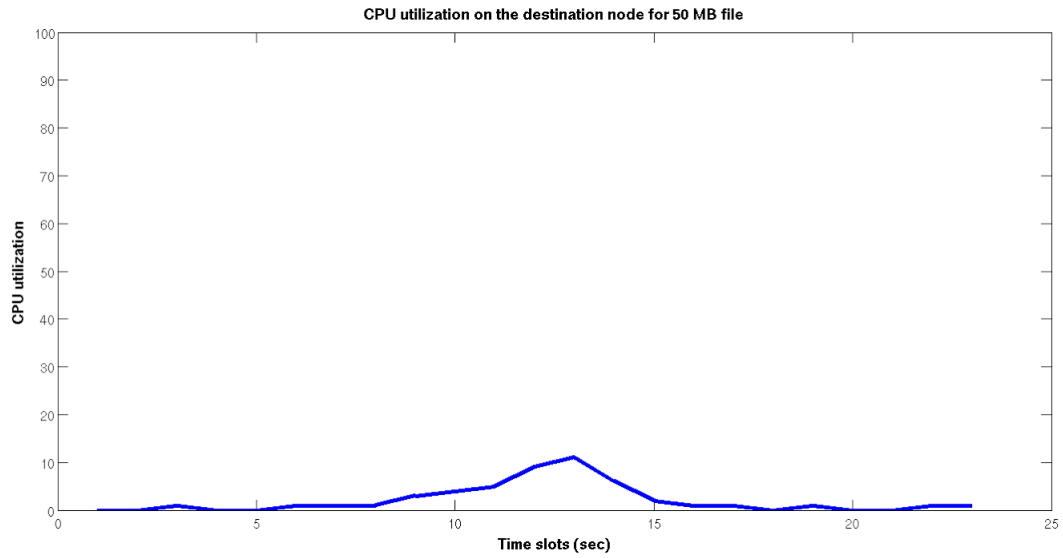

Figure 4.2: CPU utilization on the destination node while synchronizing a 50MB file.

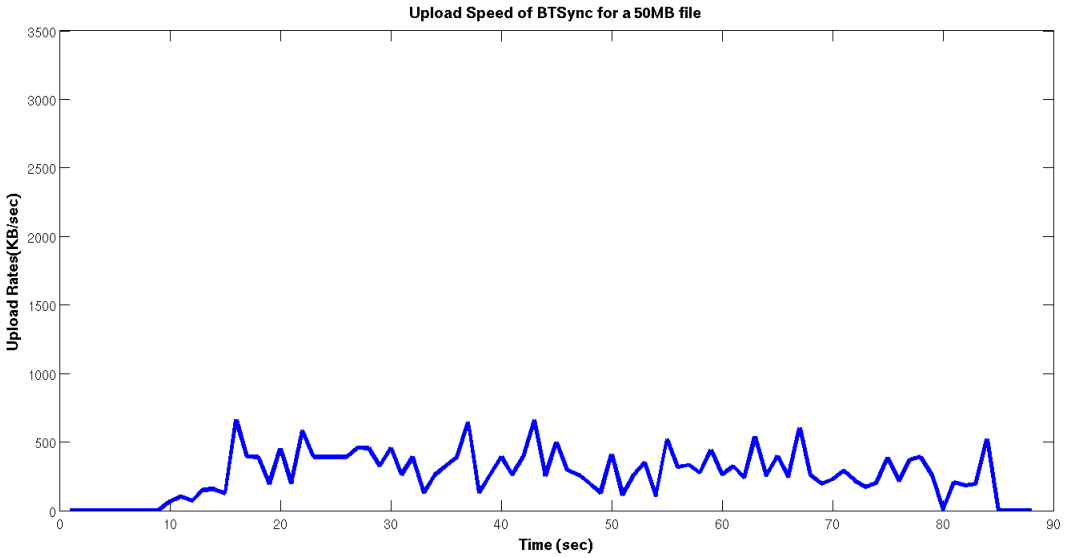

Figure 4.3: Upload speeds of a BTSync user for 50MB file.

decreases. Whereas at the destination (Figure 4.2), the CPU utilization slightly increases after a certain time and then remains constant and thereafter becomes zero when the file is completely downloaded. The CPU utilization at the destination is due to the reason that the chunks are combined to produce the original file. This stage can be considered as a

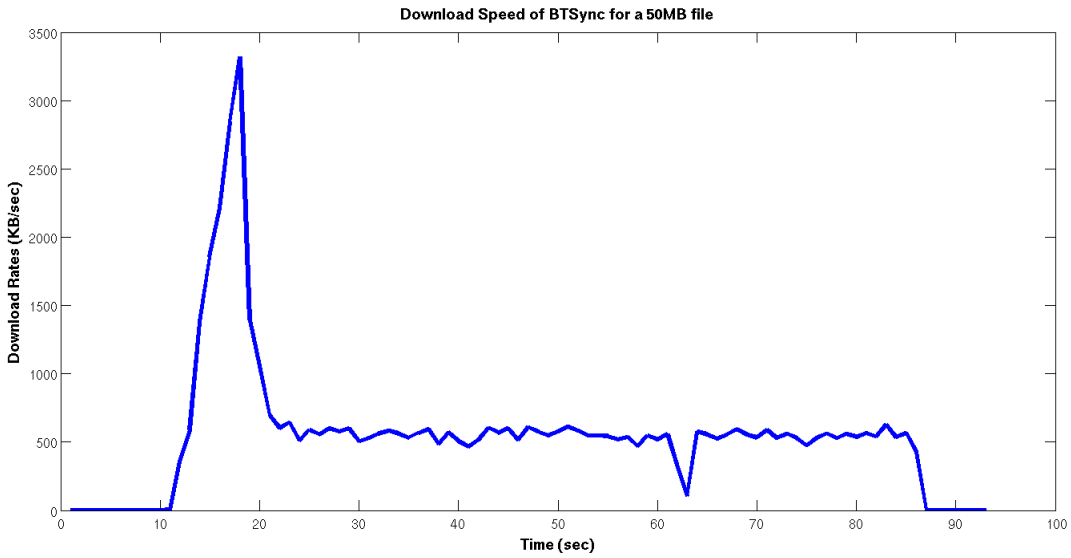

Figure 4.4: Download speeds of a BTSync user for 50MB file.

post-processing stage.

After the file is decomposed into smaller chunks, the uploading of these chunks begins at the source and hence the outgoing traffic on the data source node increases. When a chunk is uploaded, it is available for downloading at the destination and therefore, all the uploaded chunks are downloaded. The working of a P2P file synchronization protocol is different from a cloud file synchronization application where the destination node waits until the file at the source node is completely uploaded to the cloud. Therefore, in BitTorrent Sync application the uploading and downloading rates need to be captured simultaneously. Figure 4.3 and Figure 4.4 present the uploading and downloading rates on a BitTorrent Sync leecher while synchronizing a 50MB file from the seeder. Based on these two figures, we observe that there is no Tit-for-Tat protocol present in BitTorrent Sync application. The reason is that when there are low uploading speeds we observe good downloading speeds. Therefore, we conclude that there is no Tit-for-Tat protocol in BitTorrent Sync.

4.2 Understand BitTorrent Sync Performance

This section focuses on understanding the BitTorrent Sync performance by analyzing the experimental results. Three different types of experiments were conducted to understand the BTSync performance bottlenecks.

4.2.1 Experiment 1 - Measuring Synchronization Latency

We performed an experiment to find the synchronization latency of the application between two PlanetLab nodes with different file sizes such as 15 MBytes, 30 MBytes, 60 MBytes, 120 MBytes and 240 MBytes. We run this experiment 4 times and present the average synchronization latency of the application in Figure 4.5. The statistics of this experiment is presented in Table 4.1.

File Size	AVG	STD(Standard Deviation)	MAX	MIN
15 MBytes	42.7 sec	9.19	50 sec	34 sec
30 MBytes	66 sec	0.7	69 sec	62 sec
60 MBytes	131.5 sec	24.74	165 sec	92 sec
120 MBytes	209.5 sec	18.38	239 sec	178 sec
240 MBytes	375.75 sec	19.09	489 sec	271 sec

Table 4.1: Synchronization Latency with different file sizes

For examining the synchronization latency across more BitTorrent Sync users, we conducted a PlanetLab based experiment across 50 nodes using file size of 30 MBytes and 500 MBytes.

We depict from Figure 4.6 and 4.7 that out of 50 peers, 4-6 peers take more time than usual to synchronize the contents of the file using BitTorrent Sync application. From the Cumulative Distribution Function graph (Figure 4.8), for a 30 MB file 95% of the peers can download the file within 4 minutes whereas the remaining peers have to wait for more than 5 minutes. Also from Figure 4.9, for a 500MB file it is observed that 25% of the peers

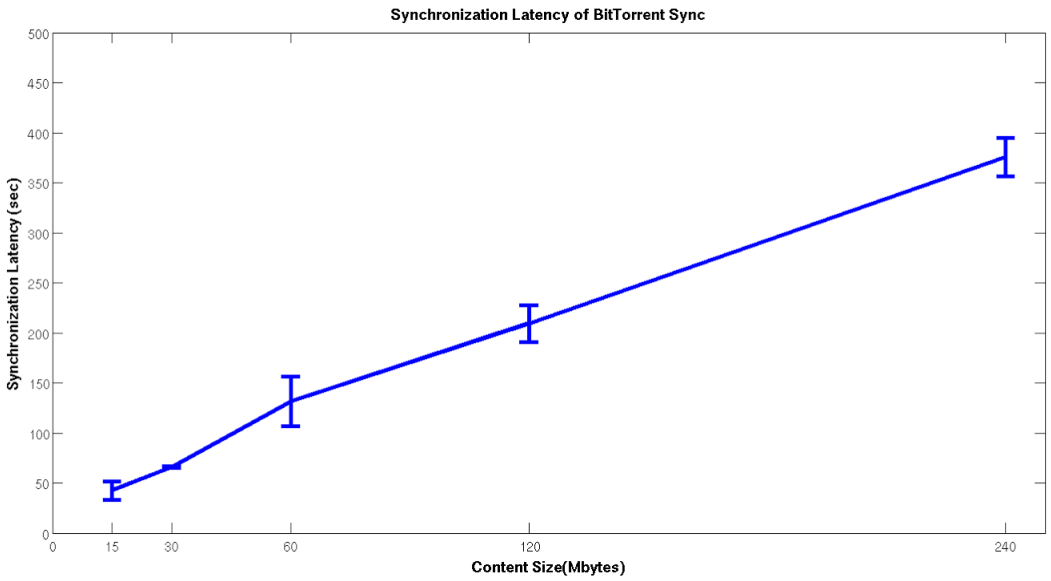

Figure 4.5: Synchronization Latency of BitTorrent Sync.

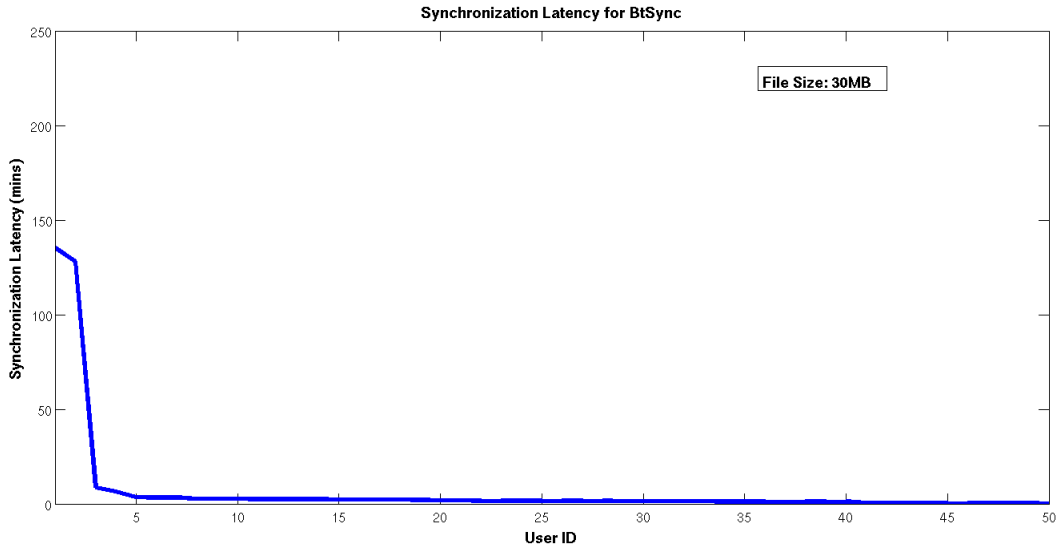

Figure 4.6: Synchronization Latency of BitTorrent Sync application for 30MB file over 50 nodes.

can download the file within 5 minutes, another 50% of the peers can download the file within 11 minutes, and the remaining 25% peers take more time to download the file.

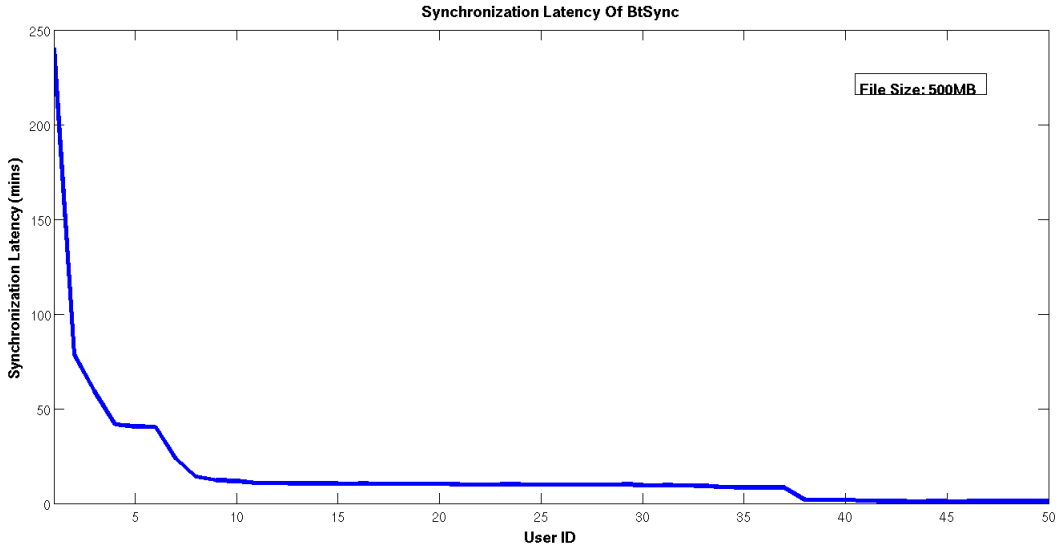

Figure 4.7: Synchronization Latency of BitTorrent Sync application for 500MB file over 50 nodes.

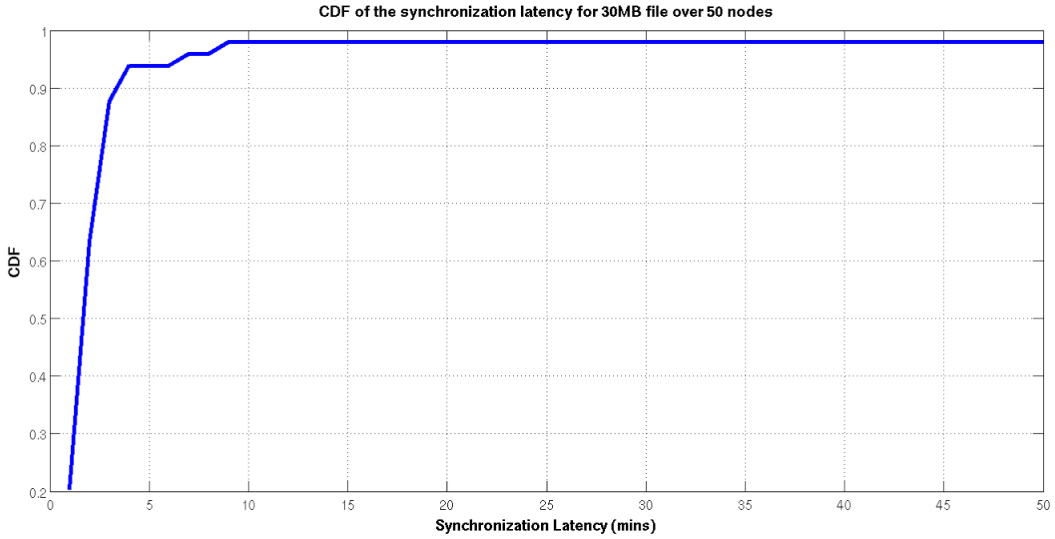

Figure 4.8: CDF of Synchronization Latency of BitTorrent Sync for 30MB file over 50 nodes.

4.2.2 Experiment 2 - Finding Uploading/Downloading Rates

We conducted another experiment to find the uploading/downloading rates of BitTorrent Sync on PlanetLab. Since BitTorrent Sync is a Peer-to-peer file synchronization pro-

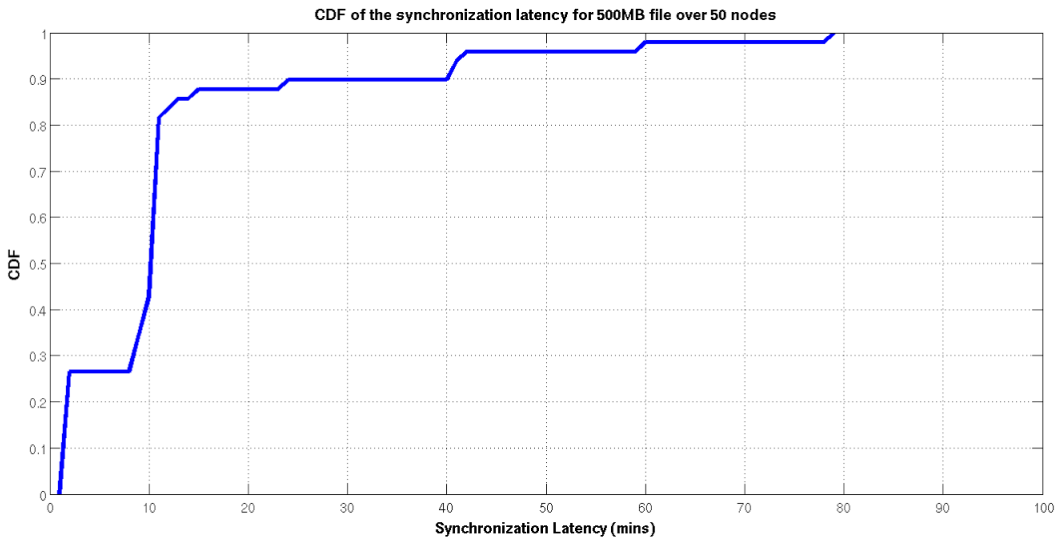

Figure 4.9: CDF of Synchronization Latency of BitTorrent Sync for 500MB file over 50 nodes.

tocol, each peer serves as both source and destination at the same time. That means, when a peer starts uploading a file to multiple peers in the network, each receiving peer downloads chunks of the file and at the same time uploads these downloaded chunks to the other peers. Hence, this experiment is conducted such that on each PlanetLab node both the uploading and downloading speeds are captured simultaneously. The files with different sizes such as 30 MB and 500 MB are automatically created by using the Linux command line utility "dd" and thereafter used for measuring the uploading/downloading speeds. Following is the example for creating a 30 MB file using "dd":

```
# dd bs=1MB count=30 if=/dev/urandom of=SpeedTestFile.txt
```

From Figure 4.10 we observe that for a 30 MB file, 40% of the peers have recorded more than 500 KBytes/sec uploading speeds. However, from Figure 4.11, we see that nearly 50% of the peers show uploading speeds of more than 1000 KBytes/sec. Therefore, from these experimental results, it can be depicted that the uploading speeds increase with the file sizes. Similarly, Figure 4.12 shows the downloading speeds of a 30 MB file on

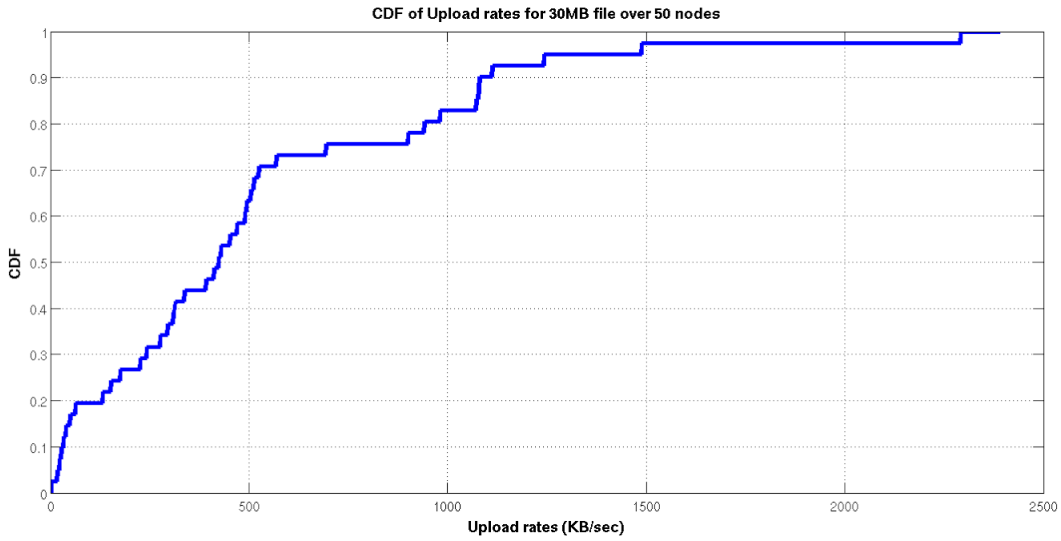

Figure 4.10: CDF of Upload Rates for 30MB file over 50 nodes.

50 nodes. From that figure, we see that 50% of the nodes have more than 700 KBytes/sec downloading speeds. Also, Figure 4.13 indicate that 55% of the nodes have downloading speeds of more than 1000 KBytes/sec. Hence, a similar type of conclusion can be drawn that the downloading speed increases with the file sizes.

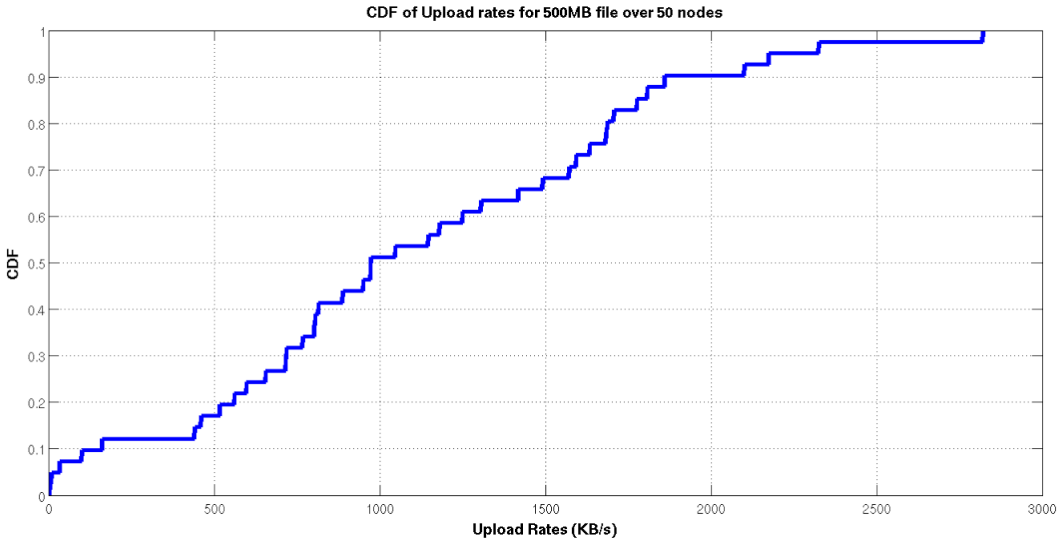

Figure 4.11: CDF of Upload Rates for 500MB file over 50 nodes.

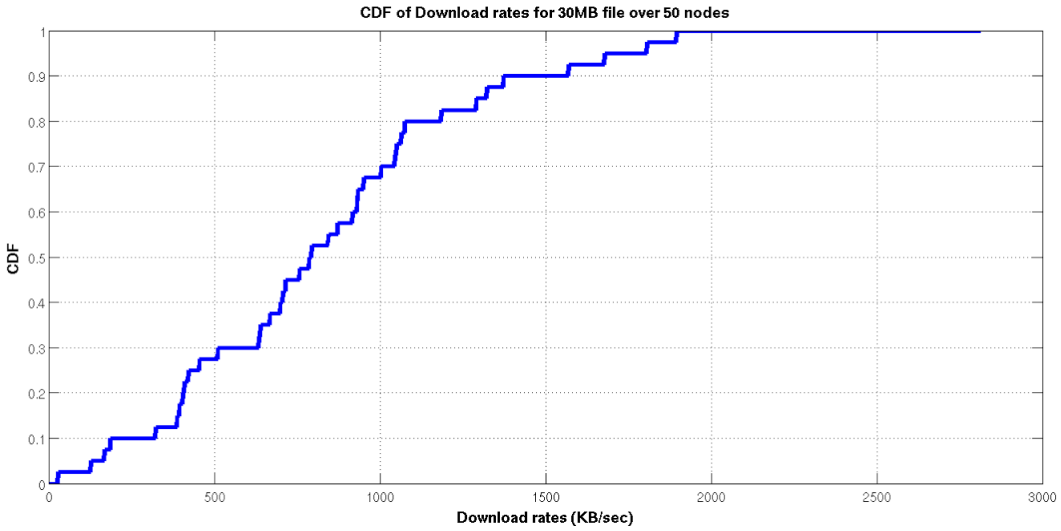

Figure 4.12: CDF of Download Rates for 30MB file over 50 nodes.

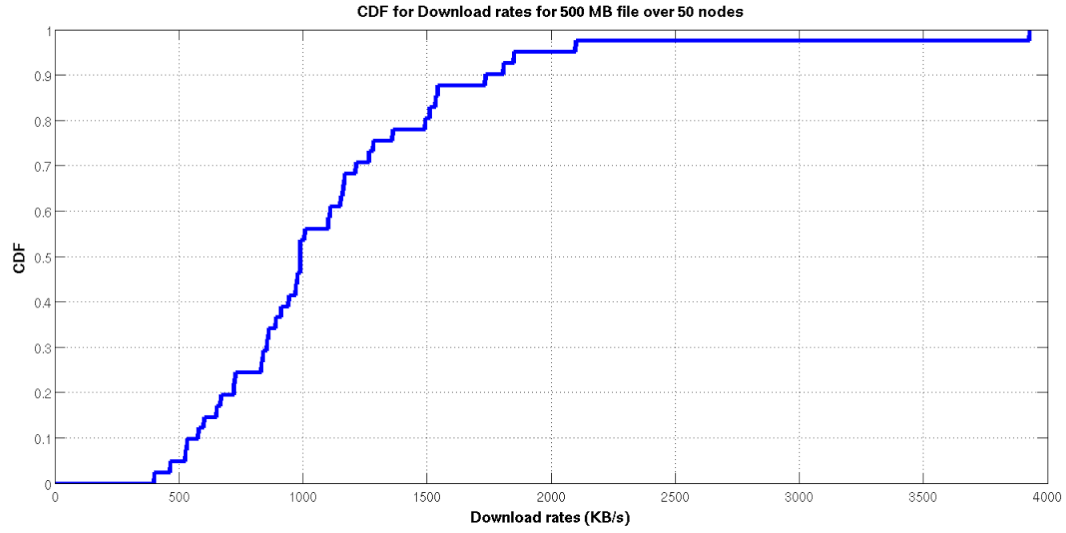

Figure 4.13: CDF of Download Rates for 500MB file over 50 nodes.

From all the ErrorBar Figures 4.14, 4.15, 4.16, 4.17, we observe that there is no guarantee of Quality of Service (QoS) for the BitTorrent Sync application because the variations in the ErrorBar plots are large.

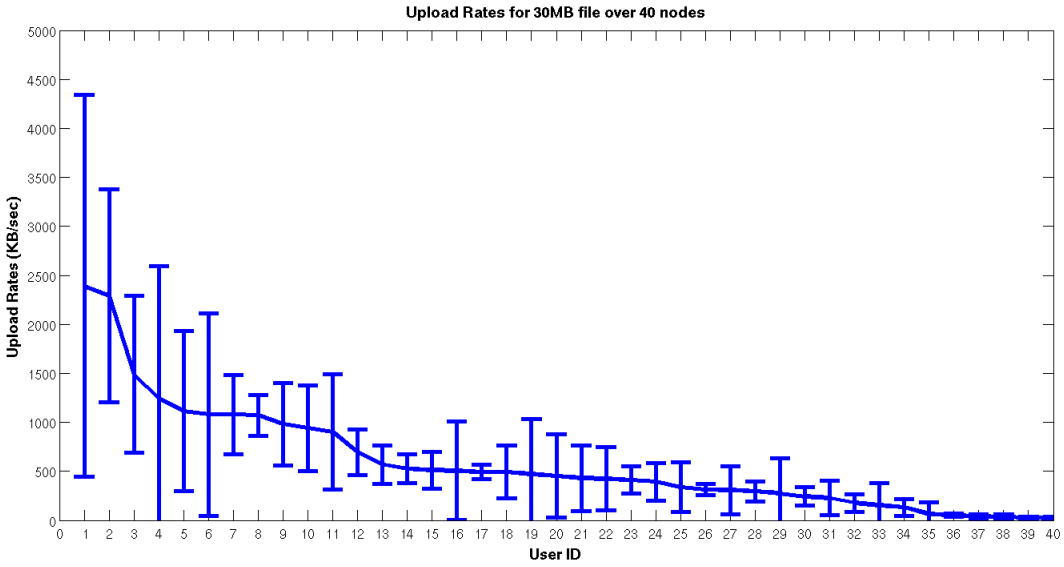

Figure 4.14: ErrorBar plot for Upload Rates for a 30MB file.

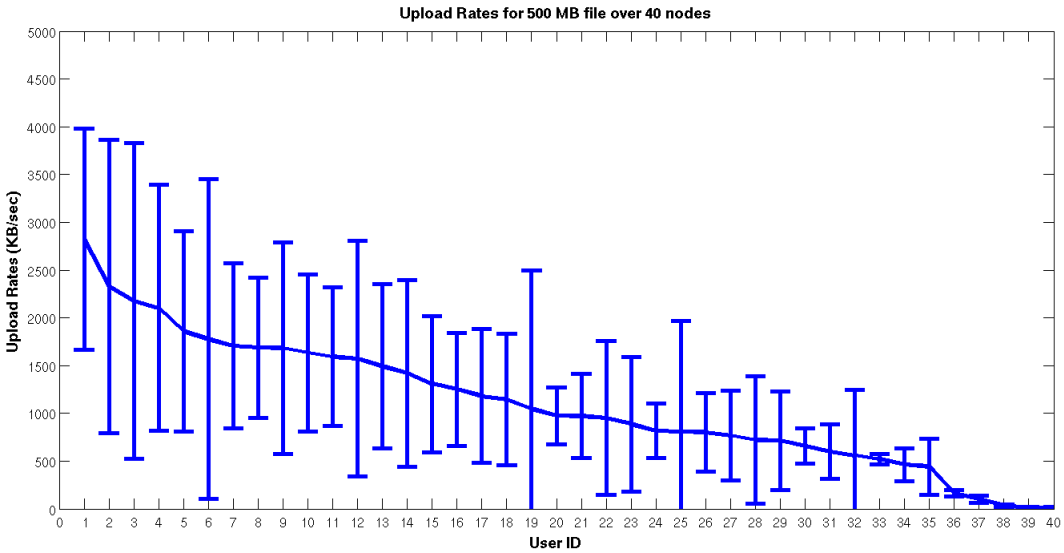

Figure 4.15: ErrorBar plot for Upload Rates for a 500MB file.

4.2.3 Experiment 3 - Measuring CPU utilization

The Cumulative Distribution Function graph of CPU utilization for a 30 MB file is presented in the Fig. 4.18. From the figure, we observe that CPU utilization of 85% of the peers does not exceed 15% while syncing a 30 MB file. Therefore, BitTorrent Sync

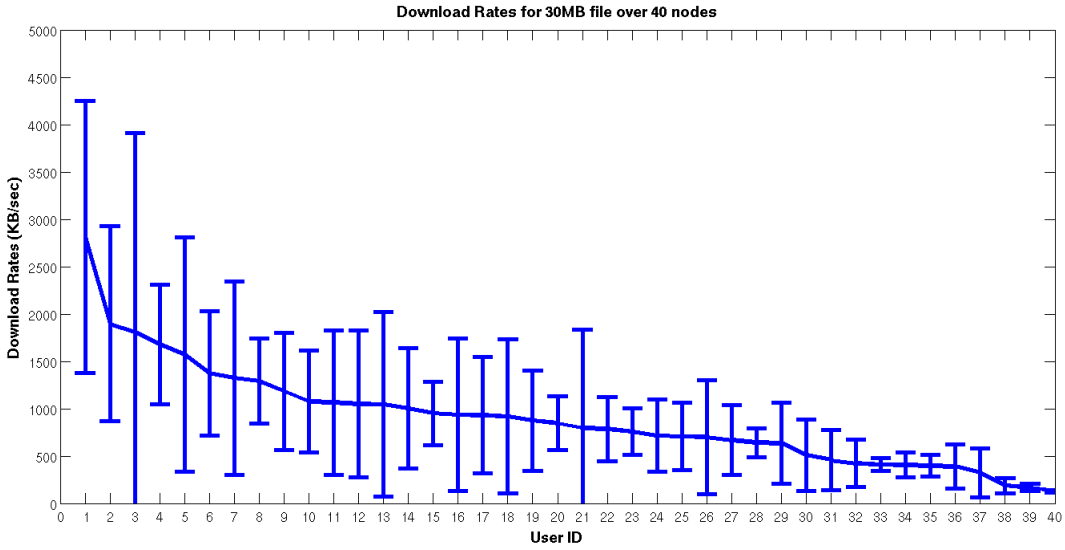

Figure 4.16: ErrorBar plot for Download Rates for a 30MB file.

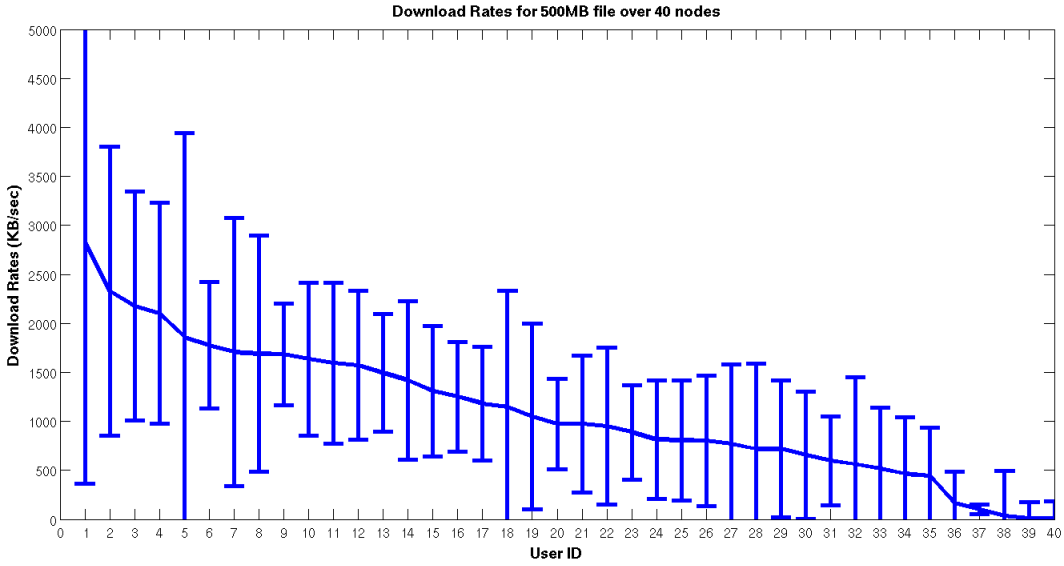

Figure 4.17: ErrorBar plot for Download Rates for a 500MB file.

application does not consume more CPU resources while synchronizing the file contents.

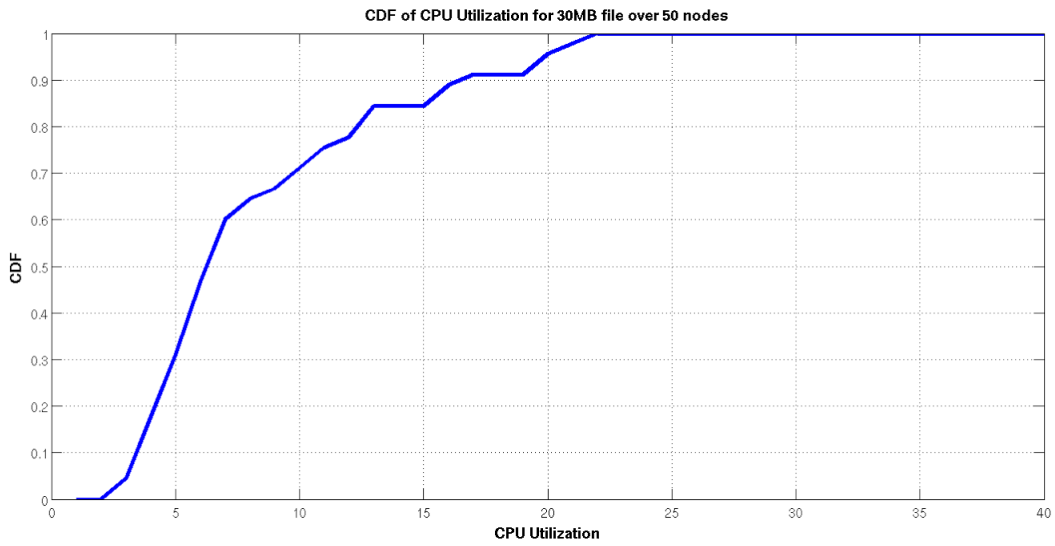

Figure 4.18: CDF of CPU utilization for 30MB file over 50 PlanetLab nodes.

4.3 Discussions

Our experimental results indicate that BitTorrent Sync file synchronization protocol has fairness issues. From Figure 4.8, 5% of the peers are taking more time to sync the 30 MB file. With the file size increased to 500MB, the peers synchronize the file very fast but 10-13% peers have more fairness problems. In our experiments, large number of peers can finish the synchronization of a 500MB file within 12 minutes while 10% of the peers may have to wait for more than 30 minutes. Therefore, the variance of the synchronization latency increases with the number of peers, leading to higher performance variance. This fairness problem may be the cause of BitTorrent Sync application itself or due to some other factors such as ISPs throttling the BitTorrent Sync traffic, firewall blocks etc.

As BitTorrent Sync inherits a lot of features from BitTorrent, same is the case with the uploading and downloading rates of BitTorrent Sync. The results also indicate that the uploading and downloading rates of BTSync increase with the increase in file size in the process of file synchronization using BitTorrent Sync. We identify that this application

does not follow Tit-for-Tat protocol of BitTorrent since there will not be any “free riding” in the case of file synchronization. From the ErrorBar graphs, we observe that there is no guarantee of QoS for BTSync because of the large variations in uploading and downloading rates.

5 Enhancing the fairness of P2P-based file synchronization: A Hybrid Cloud-P2P system

Our measurement studies have revealed the severe fairness issue in BitTorrent Sync. It is known that such a problem will largely reduce the system applicability especially for delay sensitive applications [34]. To address this problem, we explore the potential benefit of a hybrid Cloud-P2P design for file synchronization. Our main idea is to utilize the stable cloud resources to help the slowest peers in P2P synchronization system. To provide a real-system-based case study, we use BitTorrent Sync and Dropbox as the representatives of P2P and Cloud system, respectively.

5.1 Framework Design

The main focus of this hybrid framework is to mitigate the fairness problem and therefore, to improve the downloading rates of all the users. The framework design is presented in the Fig. 5.1.

5.1.1 Components in Hybrid Cloud-P2P File Synchronization

The components present in Hybrid Cloud-P2P system are discussed below:

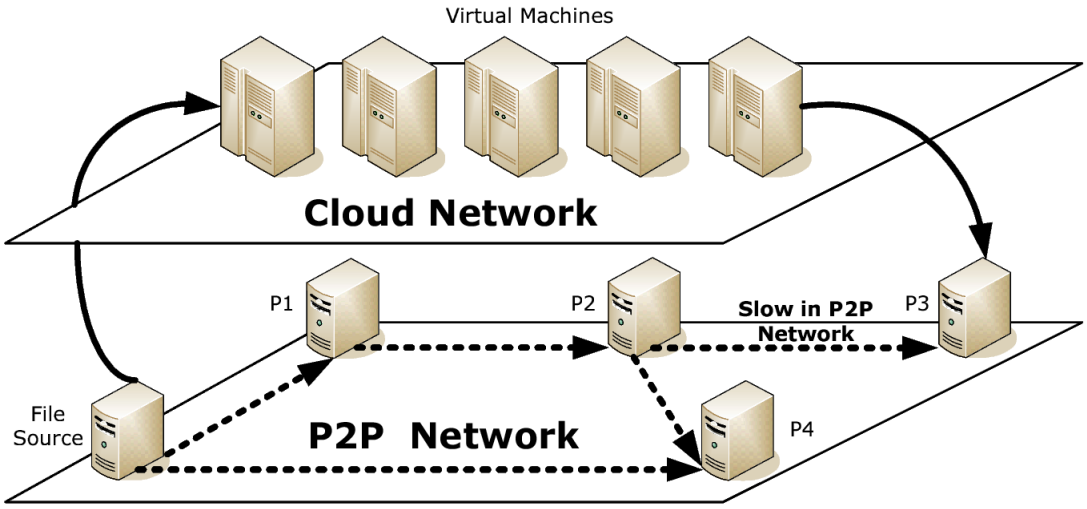

Figure 5.1: Hybrid Cloud-P2P File Synchronization.

- In the above Figure 5.1, peers P1, P2, P3 and P4 are leechers and File Source is a seeder which holds a file that is to be distributed among leechers using P2P-based file synchronization (e.g., BitTorrent Sync).
- All the peers while distributing chunks of a file among themselves using P2P file synchronization form a P2P network.
- To locate the peers in order to share the file, a tracker or Distributed Hash Table (DHT) is deployed.
- The cloud file synchronization system utilizes virtual machines to support reliable file synchronization to the end users.

5.1.2 Protocols and Interactions

The protocols and interactions between the components of Hybrid Cloud-P2P system are explained below in two stages.

Initialization Stage

1. Initially, the file source decomposes the file into multiple chunks of same size and then it uploads the chunks to remaining peers in the P2P network i.e to the leechers.

2. The file source or seeder has to generate a secret key for the file that needs to be distributed and then share it with the leechers, so that they can download the file with the help of shared secret key from seeder.

3. Additionally, the file source has to upload the file contents to the cloud so that when a slow peer is identified, it can download the content using the cloud file synchronization.

4. A tracker is used to keep the track of all the slowest peers.

5. Each peer performs a measurement to understand the cloud performance and records the download speed from the cloud. Also, a peer is to be prepared to download the file contents from the seeder using P2P file synchronization system.

Synchronization Stage

1. The download speed of each peer is compared with its previously recorded cloud download speed. If the cloud speed is greater than the speed from P2P downloading, then the peer will switch to cloud. That means, it downloads file from the cloud.

2. Upon downloading the contents from cloud, such peers help others with their P2P downloading.

To explain the functioning of Hybrid Cloud-P2P system, an example is illustrated: In the Fig. 5.1, a slow peer P3 is identified while performing the file synchronization of a 500MB file using P2P system. That identified slow peer is made to download the file from cloud to improve the fairness of P2P system and the overall system performance.

The proposed system has been implemented on PlanetLab to discover if this hybrid design can solve the fairness issue. To support this framework, Dropbox as a case study for Cloud-based file synchronization system has been deployed on all the PlanetLab nodes along with BitTorrent Sync. After deploying Dropbox on all the nodes, each node has to be linked to an existing Dropbox account to get it started. Therefore, upon linking each node gets associated to the same Dropbox account.

5.2 Performance Evaluation

To analyze the performance and fairness metrics of the proposed Hybrid Cloud-P2P file synchronization system, we conducted experiments on PlanetLab. In these experiments, even though we have used BitTorrent Sync and Dropbox as a case study, we analyze the proposed hybrid file synchronization framework as a single system. As we have discussed that, finding the slower peers in BitTorrent Sync file synchronization system is not a challenging task. Hence, our main focus is to discover the potentiality of Cloud-based file synchronization (in this case Dropbox) in accelerating the slower peers in Peer-to-peer like systems such as BitTorrent Sync from the experimental results.

5.2.1 Fairness of Hybrid Cloud-P2P

To evaluate the fairness of the proposed system, we performed an experiment to find the downloading rates on 50 PlanetLab nodes. Out of these 50 nodes, 40 nodes run on

BitTorrent Sync application and the remaining nodes run on Dropbox. This experiment is carried out for synchronizing a 500MB file. The results are plotted in the Fig. 5.2. From this figure, we notice that all the nodes have promising downloading rates. In addition, from Fig. 5.3 we see that all the nodes download the 500MB file within 15 minutes. Therefore, these experiments indicate that the Dropbox nodes help to improve the fairness of the entire system.

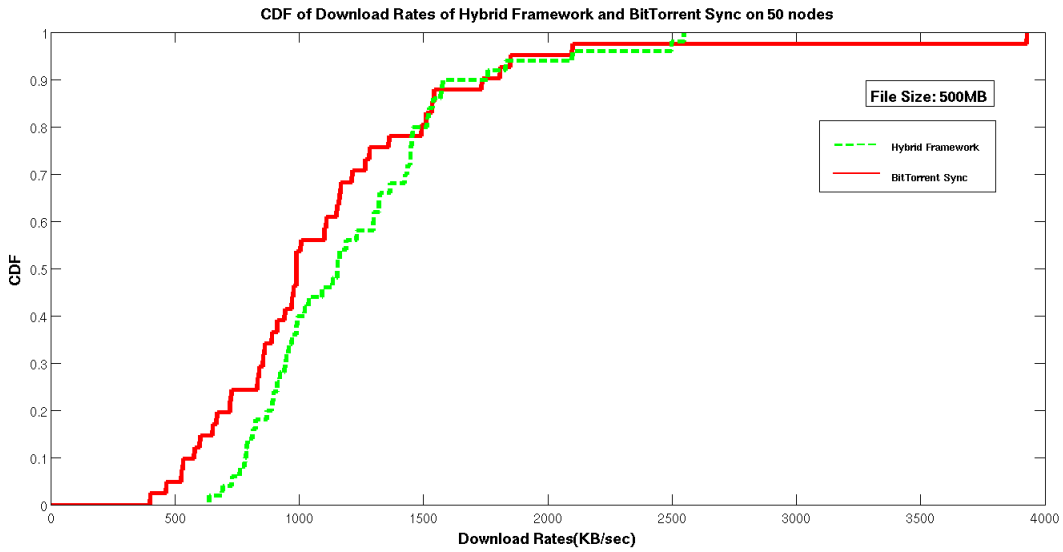

Figure 5.2: CDF of Download Rates of both Hybrid Framework and BitTorrent Sync on 50 nodes.

5.2.2 Performance of Hybrid Cloud-P2P

To understand the performance of Hybrid Cloud-P2P system, we demonstrate an experiment by synchronizing the 500MB file on 50 nodes. From the Fig. 5.4, 80% of the users downloads the file within 12 minutes and the remaining 20% downloads in 16 minutes. Hence, Dropbox helps the slower peers to improve their downloading speeds and thereby reduces the synchronization latency. The evaluation shows that the cloud-based enhancement can well-address the fairness and performance issues while improving the overall

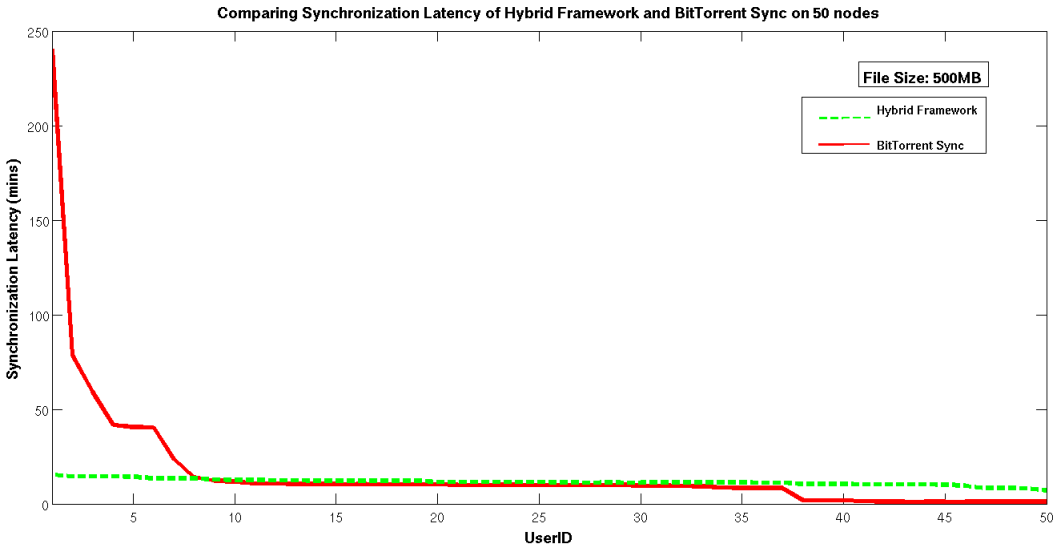

Figure 5.3: Synchronization Latency of Hybrid Framework and BitTorrent Sync on 50 nodes.

synchronization efficiency by 38%.

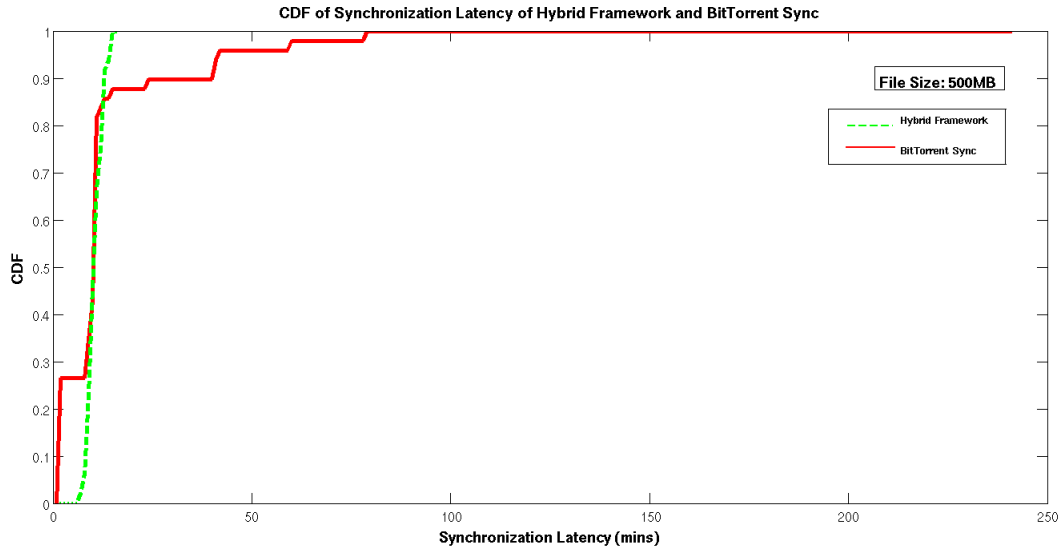

Figure 5.4: CDF of Synchronization Latency of both Hybrid Framework and BitTorrent Sync on 50 nodes.

5.3 Further Discussions

There have been some practical issues while implementing the proposed Hybrid Cloud-P2P system. Even though the overall performance can be improved, a user has to use both Peer-to-Peer and cloud-based file synchronization applications. That is, to synchronize a file a user has to rely on two kinds of file synchronization applications which is not an effective approach. Moreover, if a user needs to download large contents, he/she has to pay the expenses for hosting files on cloud. Furthermore, in the Hybrid framework slower peers of Peer-to-peer file synchronization application are replaced by cloud-based file synchronization systems. But it is not exactly clear which peers are to be considered as slower peers so that they can do the file synchronization with a cloud-based system.

6 Conclusion and Future Work

In this thesis, we for the first time examined the performance of peer-to-peer file synchronization in real-world measurement. We also presented the challenges involved in performing these measurements on highly distributed PlanetLab testbed. Our experimental results showed that the P2P file synchronization system can provide very efficient file synchronization especially for large contents. However, our PlanetLab experiments indicate that BitTorrent Sync suffers from fairness and performance issues. To alleviate these problems, we proposed a hybrid cloud-P2P system by merging the P2P and cloud-based file synchronization systems. The evaluation of this hybrid cloud-P2P system shows that it can well address the fairness issues while improving the overall synchronization efficiency. Our insights into not only the general design tradeoffs of Hybrid Cloud-P2P system but also the practical implementation experiences will be helpful to guide future designs of Cloud-P2P file synchronization systems for a variety of purposes. There are many possible future avenues, and we are particularly interested in the following critical issues:

- More studies are required to decide about the number of slower peers in Peer-to-peer file synchronization that needs to be replaced by the cloud file synchronization systems to achieve efficient synchronization.

- When a slow peer is found, the cloud-based file synchronization is triggered on that peer. By that time, a certain number of chunks are already downloaded by that peer. Instead of starting the file download from the scratch again, further studies are necessary in order to download only the remaining chunks using the cloud-based file

synchronization system which largely improves the synchronization latency of that peer.

Bibliography

[1] R. M. Adler. ``Distributed coordination models for client/server computing". In: *Computer* 28.4 (1995), pp. 14–22 (cit. on p. 4).

[2] S. Androutsellis-Theotokis and D. Spinellis. ``A survey of peer-to-peer content distribution technologies". In: *ACM Computing Surveys (CSUR)* 36.4 (2004), pp. 335–371 (cit. on p. 11).

[3] R. Bindal, P. Cao, W. Chan, J. Medved, G. Suwala, T. Bates, and A. Zhang. ``Improving traffic locality in BitTorrent via biased neighbor selection". In: *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*. IEEE. 2006, pp. 66–66 (cit. on pp. 9, 10, 12).

[4] *BitTorrent Sync Homepage*. https://www.getsync.com/. Accessed: 2015-07-20 (cit. on pp. 20, 22, 24).

[5] K. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. ``Enabling dynamic content caching for database-driven web sites". In: *ACM SIGMOD Record*. Vol. 30. 2. ACM. 2001, pp. 532–543 (cit. on p. 5).

[6] C. Carbunaru, Y. M. Teo, B. Leong, and T. Ho. ``Modeling flash crowd performance in peer-to-peer file distribution". In: *Parallel and Distributed Systems, IEEE Transactions on* 25.10 (2014), pp. 2617–2626 (cit. on p. 12).

[7] P. Casas, H. R. Fischer, S. Suette, and R. Schatz. ``A first look at quality of experience in personal cloud storage services". In: *Communications Workshops (ICC), 2013 IEEE International Conference on*. IEEE. 2013, pp. 733–737 (cit. on p. 18).

[8] B. Cohen. ``Incentives build robustness in BitTorrent". In: *Workshop on Economics of Peer-to-Peer systems*. Vol. 6. 2003, pp. 68–72 (cit. on p. 6).

[9] R. Cuevas, M. Kryczka, A. Cuevas, S. Kaune, C. Guerrero, and R. Rejaie. ``Is content publishing in BitTorrent altruistic or profit-driven?" In: *Proceedings of the 6th International COnference*. ACM. 2010, p. 11 (cit. on p. 8).

[10] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras. ``Inside dropbox: understanding personal cloud storage services". In: *Proceedings of the 2012 ACM conference on Internet measurement conference*. ACM. 2012, pp. 481–494 (cit. on pp. 1, 17).

[11] *Dropbox Homepage*. https://www.dropbox.com/. Accessed: 2015-07-23 (cit. on p. 1).

[12] J. Farina, M. Scanlon, and M.-T. Kechadi. ``BitTorrent Sync: first impressions and digital forensic implications". In: *Digital Investigation* 11 (2014), S77–S86 (cit. on p. 20).

[13] A. Gandhi, M. Harchol-Balter, and I. Adan. ``Server farms with setup costs". In: *Performance Evaluation* 67.11 (2010), pp. 1123–1138 (cit. on p. 5).

[14] G. Goncalves, I. Drago, A. P. Couto da Silva, A. Borges Vieira, and J. M. Almeida. ``Modeling the dropbox client behavior". In: *Communications (ICC), 2014 IEEE International Conference on*. IEEE. 2014, pp. 1332–1337 (cit. on p. 17).

[15] *Google Drive Homepage*. https://www.google.com/drive/. Accessed: 2015-07-23 (cit. on p. 1).

[16] R. Gracia-Tinedo, M. Sanchez Artigas, A. Moreno-Martinez, C. Cotes, and P. Garcia Lopez. ``Actively measuring personal cloud storage". In: *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*. IEEE. 2013, pp. 301–308 (cit. on p. 17).

[17] W. Hu, T. Yang, and J. N. Matthews. ``The good, the bad and the ugly of consumer cloud storage". In: *ACM SIGOPS Operating Systems Review* 44.3 (2010), pp. 110–115 (cit. on p. 18).

[18] J. Jing, A. S. Helal, and A. Elmagarmid. ``Client-server computing in mobile environments". In: *ACM computing surveys (CSUR)* 31.2 (1999), pp. 117–157 (cit. on p. 4).

[19] M. Kryczka, R. Cuevas, C. Guerrero, A. Azcorra, and A. Cuevas. ``Measuring the bittorrent ecosystem: Techniques, tips, and tricks". In: *Communications Magazine, IEEE* 49.9 (2011), pp. 144–152 (cit. on pp. 7, 11).

[20] Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu, L. Cheng, Y. Liu, Y. Dai, and Z.-L. Zhang. ``Towards network-level efficiency for cloud storage services". In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM. 2014, pp. 115–128 (cit. on pp. 1, 13, 14).

[21] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Y. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai. ``Efficient batched synchronization in dropbox-like cloud storage services". In: *Middleware 2013*. Springer, 2013, pp. 307–327 (cit. on p. 1).

[22] N. Magharei and R. Rejaie. ``Prime: Peer-to-peer receiver-driven mesh-based streaming". In: *IEEE/ACM Transactions on Networking (TON)* 17.4 (2009), pp. 1052–1065 (cit. on p. 12).

[23] *Microsoft OneDrive file sync problems*. https://www.kuppingercole.com/blog/small/microsoft-onedrive-file-sync-problems. Accessed: 2015-07-20 (cit. on p. 18).

[24] *Microsoft OneDrive Homepage*. https://onedrive.live.com/. Accessed: 2015-07-23 (cit. on p. 1).

[25] A.-M. K. Pathan and R. Buyya. ``A taxonomy and survey of content delivery networks". In: *Grid Computing and Distributed Systems Laboratory, University of Melbourne, Technical Report* (2007) (cit. on p. 4).

[26] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. ``Load balancing and unbalancing for power and performance in cluster-based systems". In: *Workshop on compilers and operating systems for low power*. Vol. 180. Barcelona, Spain. 2001, pp. 182–195 (cit. on p. 5).

[27] *PlanetLab Bandwidth Limitations*. https://www.planet-lab.org/doc/BandwidthLimits. Accessed: 2015-07-20 (cit. on p. 40).

[28] *PlanetLab Homepage*. http://planet-lab.org/. Accessed: 2015-07-20 (cit. on p. 27).

[29] J. A. Pouwelse, P. Garbacki, D. Epema, and H. Sips. *A measurement study of the bittorrent peer-to-peer file-sharing system*. Tech. rep. Technical Report PDS-2004-003, Delft University of Technology, The Netherlands, 2004 (cit. on p. 6).

[30] *Python Global Interpreter Lock*. https://docs.python.org/2/glossary.html/#term-global-interpreter-lock. Accessed: 2015-07-21 (cit. on p. 40).

[31] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. ``Improving datacenter performance and robustness with multipath tcp". In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 266–277 (cit. on p. 4).

[32] *Rsync Homepage*. https://rsync.samba.org/. Accessed: 2015-07-21 (cit. on p. 32).

[33] M. Scanlon, J. Farina, and M. Kechadi. ``Bittorrent sync: Network investigation methodology". In: *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*. IEEE. 2014, pp. 21–29 (cit. on pp. 19, 22).

[34] *Twitter Uses BitTorrent for Server Deployment*. https://torrentfreak.com/twitter-uses-bittorrent-for-server-deployment-100210/. Accessed: 2015-07-22 (cit. on p. 55).

[35] E. M. Van Eenennaam. ``Performance Factors in peertopeer File Distribution Networks". In: () (cit. on p. 5).

[36] *Vxargs Homepage*. http://vxargs.sourceforge.net/. Accessed: 2015-07-20 (cit. on p. 31).

[37] H. Wang, J. Liu, and K. Xu. ``Exploring BitTorrent peer distribution via hybrid PlanetLab-Internet measurement". In: *Quality of Service (IWQoS), 2010 18th International Workshop on*. IEEE. 2010, pp. 1–2 (cit. on p. 11).

[38] H. Wang, R. Shea, F. Wang, and J. Liu. ``On the impact of virtualization on dropbox-like cloud file storage/synchronization services". In: *Proceedings of the 2012 IEEE 20th international workshop on quality of service*. IEEE Press. 2012, p. 11 (cit. on p. 17).

[39] *Who's winning the consumer cloud storage wars?* http://fortune.com/2014/11/06/dropbox-google-drive-microsoft-onedrive/. Accessed: 2015-07-20 (cit. on p. 14).

[40] Wikipedia. *Comparison of BitTorrent clients --- Wikipedia, The Free Encyclopedia*. [Online; accessed 20-July-2015]. 2015. URL: https://en.wikipedia.org/w/index.php?title=Comparison_of_BitTorrent_clients&oldid=668689477 (cit. on p. 8).

[41] R. L. Xia and J. K. Muppala. ``A survey of BitTorrent performance". In: *Communications Surveys & Tutorials, IEEE* 12.2 (2010), pp. 140–158 (cit. on p. 6).

[42] Y. Zhang, C. Dragga, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. ``*-Box: towards reliability and consistency in dropbox-like file synchronization services". In: *Proceedings of the 5th USENIX conference on Hot Topics in Storage and File Systems*. USENIX Association. 2013, pp. 2–2 (cit. on p. 15).